SYSTEMS AND METHODS FOR PREVENTING UNAUTHORIZED USE OF DIGITAL CONTENT

5 **RELATED APPLICATIONS**

This application is a continuation-in-part application of United States Patent Application Number 09/960,610, filed September 21, 2001, which application claims the benefit of United States Provisional Application Serial No. 60/234,657, filed September 22, 2000, United States Provisional Application Serial No. 60/240,611, filed October 16, 2000, United States Provisional

10 Application Serial No. 60/242,949, filed October 24, 2000, and United States Provisional Application Serial No. 60/244,704, filed October 31, 2000.

This application also claims the benefit of United States Provisional Application Serial No. 60/249,946, filed November 20, 2000, United States Provisional Application Serial No. 60/260,705, filed January 10, 2001, and United States Provisional Application Serial No.

15 60/285,300, filed April 20, 2001.

The contents of the applications referenced above are incorporated herein by reference, in their entirety.

BACKGROUND OF THE INVENTION

20

Field of the Invention

This invention is related to the field of protecting digital information from being copied, modified, or used by unauthorized parties. In particular this invention is related to systems and methods that prevent unauthorized access to, and modification of, digital data as found on

25 computer systems and consumer-appliance systems that utilize Compact Disc (CD), DVD, or other removable media (such as Flash Memory on standard or proprietary cards or sticks, or other non-volatile memory) technologies, or any storage media of any type, or any such content delivered via any network connection of any type.

Description of the Related Art

1

The electronic publishing industry for application software, computer games, appliance-console games, movies, and music, is facing a growing and serious problem; namely, the piracy and unauthorized modification and use of their content. Since digital content is by nature capable of being copied exactly, wherein a copy is identical in every way to the original, and since the tools to do so are increasingly available, the industry is facing increasing losses. Such losses may include the unauthorized copying of a CD containing a game, or the unauthorized reverse engineering and modification of a word processing program to allow for its illegal distribution, or the reverse engineering of a copy protection scheme to disable it, making it possible to make duplicates with ease.

There are many mechanisms available that may be used to limit or prevent unauthorized access to digital content. Following deployment, such mechanisms are often times subsequently compromised by hackers, and the methods and techniques used to compromise them have been widely disseminated and actively used and enhanced. Most protections are simplistic in nature, and depend to large degree on the secrecy of the simple method as much as its inherent security or ingenuity, such that if not defeated prior to publication, the act of publishing them, for example in patent form, reveals enough about them to render them less effective. More than one of these approaches may be defeated if anticipated by using "ProcDump", a memory lifting tool that is available free on the World Wide Web (such a tool may also be easily written following technical instructions that may also be found on the web) in conjunction with SoftICE, a powerful debugging tool, which may also be found on the web. A computer system is usually the platform and tool of choice for one intent on reverse engineering or cracking these protection mechanisms; even if the protected content's target was not a computer system such as a PC but rather an appliance computing device such as a game console, the content can best be modified ("hacked") on a computer. In terms of protecting content from copying or modification by a skilled person with a modern computer system, most inventions in the field (see below) are not protected from being reverse engineered, modified, or content-duplicated by means of commonly available tools such as "SoftICE" (an in-circuit emulator and very powerful debugger), "ProcDump" (can capture any data content from any memory location, regardless of how protected the memory was thought to be), "IDA" (a disassembler), and "FileMon" (a file system

monitoring and transcribing service tool). There are no design secrets that can be kept from such a set of tools, and there are many more such tools in existence, and more being created all the time. Therefore it becomes far more important to have well designed mechanisms that do not depend on their secrecy, as much as their design, to ensure security.

5

Many of these mechanisms depend to a great extent on lack of knowledge about the mechanisms by the persons attempting to modify or copy the content. With even partial knowledge, many of these mechanisms can be defeated by even a moderately technical person with access to the web where all the necessary tools and techniques are available. There is a need

10 for security methods that do not depend solely upon their secrecy or obscurity in order to be effective.

Summary of the Invention

To address the limitations of the conventional approaches described above, the present invention is directed to a digital content security method and system that does not depend solely

15 upon secrecy or obscurity in order to be effective.

In one aspect, the present invention is directed to a system and method for storing encrypted data, subdivided into arbitrarily small collections of bits within other files, or between them, or outside a file system's known storage areas entirely. The data size used in the discussion

20 below is 4-bit nibbles and 8-bit bytes , but it should be noted that any data size is applicable to the principles of the present invention. The location for the information is arrived at algorithmically, and no single individual location is inherently secret, but knowledge of the totality of the locations and their order of traversal is critical. The content is encrypted, but

25 before being encrypted, each 8-bit word or byte is broken down into 4-bit nibbles, and is merged 4 bits at a time with a completely unrelated stream of bits, which may also themselves be equally meaningful 4-bit nibbles. Such interleaved multiplexing is not limited to the two-way example above, but may be considered N-way, where N is an arbitrary positive integer of any size.

3

In another aspect of the present invention, the locations are not dynamically arrived at but are rather chosen by a mapping process and an encoded location map is generated. This map may be itself encrypted, then subdivided into 4-bit nibbles or 8-bit bytes and itself hidden.

5   In another aspect of the present invention, any encrypted file is locked by taking its decryption key and then encrypting that key using another encryption method or key. The encrypted key is placed in a known location, such as the beginning, end, or at a known offset within the file, or is subdivided into bits and scattered into the file in known, and therefore retrievable, locations. The locked file itself may then be subdivided, multiplexed, further

10   encrypted, and hidden, as needed.

In another aspect of the present invention, content can be replaced with translocated content, such that, in the example of executable content, the file a.exe is replaced with another file a.exe. The contents of a.exe are encrypted, locked, and hidden as described above. Upon

15   execution of a.exe the content is retrieved, decrypted if necessary, executed as desired. This is not to imply a limitation to executable software content such as .exe files; all other digital content, such as an audio a.wav file, can have one or more associations in preference order, with execution environments such as a variety of MP3 or audio software players. The playback environment can be provided within the secured entity, or can be something that was always

20   resident on the system prior to installation of the secured entity.

In another aspect of the present invention, digital content (whether or not it is also hidden and/or encrypted) is modified such that it is tokenized or otherwise obfuscated, and then when it comes time for the content to be used, it is interpreted within a custom interpreter that is a part of

25   the system. An example of such is to modify a compiler such that the assembly language output is nonstandard, and thus require that the execution occur in an interpreter designed for the task. Such construction is possible even using decades-old utilities such as LEXX and YaCC, traditionally compiler creation tools. Such an interpreter is composed of a parser which consumes tokens, converts the tokenized logic to native computing instructions, obfuscates these

30   instructions with anti-disassembly logic, and feeds them to the standard system interfaces. Such

interposition of execution layers makes debugging a nontrivial task, and the anti-disassembly logic eliminates the use of many popular disassembly tools

In another aspect, the present invention employs saturation "chaff" logic to create a large amount of harmless and meaningless (yet utterly real in appearance and content, and apparently meaningful) information designed to saturate or confuse logging, reverse engineering, and debugging tools. Such logic can be targeted at specific systems, such that large amounts of I/O to the CD device can be used to mask any meaningful activity that may also be occurring on a device. The saturation invention is particularly useful against attempts to reverse engineer a protection system by monitoring its activity, because any such eventual logging/journal output of these tools must be reviewed and interpreted by human beings,   and the overall volume (instead of 100 or 500 lines of logging on a device in a few minutes, this invention can generate tens of thousands of spurious log events in the same time period) can make it difficult or impossible to sort out the useful information from the chaff.

In another aspect, the present invention prevents sophisticated monitoring tools from monitoring and logging file access. This is accomplished by creating a driver extension layer, referred to as a "shim", and attaching it to all appropriate operating system interfaces. Note that these shim interfaces on most consumer computer operating systems allow chaining, so that multiple layers can be stacked dynamically. This is also commonly called "hooking" on Windows operating systems. The present invention provides security by selecting where to hook (whether you choose to hook before or after a monitoring shim/hooking tool, such as FileMon, is significant; one can even hook both before AND after, to provide the tool with spurious input information). The mechanism rehooks at the desired depth(s) with variable frequency to defeat subsequent monitoring tool invocations.

In another aspect the present invention creates a driver extension layer, and shims or hooks the all relevant operating system interfaces, (and re-attach as above if desired). In this aspect, access filtering capabilities are employed to alter access to secured content, or to security-threat content.

In another aspect, the present invention employs an authorization process, which serves as a significant part of the decision in determining the status and origins of a task or process on the system and make an access determination.

5

In another aspect, the present invention includes an "assassin" construct; a system entity that operates to monitor activity and take action as needed. If, for example, the system were composed of multiple processes, one or more of which were protective by nature, and someone were to kill or stop one of the protective processes, an assassin process would take note of that

10   occurrence, and would take action. The authorization process described below is a significant part of this decision in determining the status and origins of a task or process on the system. Such action might include disabling the rest of the system to prevent tampering, or killing the tampering process, or both. Assassin constructs are most useful if they serve some other purpose essential to the system, such as if, in the example above, the assassin process also served as a

15   system's decryption service, such that killing the assassin would result in loss of ability to decrypt by the system, guaranteeing failure. Such assassin processes can detect the existence of specific tools both dormant and active, and prohibit the protective system's exposure to them.

In another aspect, the present invention includes an "authorization" construct. Such a

20   process is aware of how the operating system tracks the lineage of processes and tasks, and can determine parentage quickly and accurately, so that is can be used to authorize file accesses to appropriate subtasks of an authorized task. On many operating systems the level of identification required by the system is insufficient so this aspect of the invention can bypass system query utilities and instead walk the system's process memory and track the lineage, creation, and

25   deletion of processes and tasks.

In view of the above, the present invention is first directed to a system and method for preventing unauthorized use of digital content data. Digital content data is subdivided into data segments. The data segments are modified with second data to generate modified data. The

30   modified data are then stored at predetermined memory locations.

It is noted that the digital content data may comprise any form of digital data that is stored, transmitted, or utilized on or between computer systems of all types. Such data includes, but is not limited to, audio, video, documents, electronic text and software and the like.

5

The data segments are preferably of a variable length, and the second data preferably comprises a randomly generated data stream. The second data may optionally comprise portions of the digital content data.

10

The modified data may likewise be encrypted and stored, for example with an encryption key, which, may in turn itself be encrypted. The encryption key may be stored with the encrypted modified data at the predetermined memory locations, and may be partitioned among the encrypted modified data.

15

The digital content data may comprise first and second digital content data, wherein the predetermined memory locations are selected as combinations of the locations at which the first and second digital content data were originally stored. A map of locations at which the modified data is stored may be generated and stored at the predetermined memory locations.

20

In a preferred embodiment, the memory locations reside on a system and the system is scanned to determine available memory locations. Target memory locations within the available memory locations at which to store the modified data are determined. The modified data is then stored at the target memory locations. The available memory locations may be located within file system locations and outside file system locations.

25

Modification of the data segments preferably comprises interleaving the data segments with the second data to generate interleaved data. The second data may be tokenized, for example with lexical equivalents of assembly language commands. The lexical equivalents may be consumed by a system interpreter, in turn generating alternative assembly language

30 commands selected to obfuscate the digital content data in the event of an unauthorized access.

7

The present invention is also directed to a method and system for preventing unauthorized use of digital content data in a system having memory locations comprising. Digital content data is subdivided into data segments, which are, in turn, modified with second

5   data to generate modified data. The system is scanned to determine available memory locations and target memory locations within the available memory locations at which to store the modified data are selected. The modified data are then stored at the target memory locations.

The present invention is further directed to a method and system for preventing

10   unauthorized use of digital content data hosted on a system. Digital content data is modified with saturation data to generate modified data, and the modified data are stored at predetermined memory locations on the system to deter unauthorized access of the digital content data.

In a preferred embodiment, it is determined whether an unauthorized attempt at accessing

15   the digital content data occurs, and in the event of unauthorized access, saturation traffic is generated on the system to deter the unauthorized activity. The saturation traffic may comprise commands that burden system resources, for example as a function of activity utilizing the system resources subject to the unauthorized access.

20   The present invention is further directed to a method and system for preventing unauthorized use of digital content data hosted on a system wherein a table of contents identifies files stored at memory locations of the system. A first memory location referring to a location at which at which first data file is stored is identified at the table of contents. The first memory location in the table of contents is then modified to refer to a second data file at a second

25   location. Upon an attempt at access by the system of the first data file, the second data file is accessed if the attempt is unauthorized.

In an alternative embodiment, the first data file is replaced with the second data file and upon an attempt at access by the system of the first data file, the second data file is accessed if

30   the attempt is unauthorized.

8

The present invention is further directed to a method and system for preventing unauthorized use of digital content data hosted on a system. An operating system interface of the system is monitored to determine access of operating system resources. A shim is repeatedly generated on the operating system interface to deter unauthorized access of the digital content data.

The present invention is further directed to a method and system for preventing unauthorized use of digital content data hosted on a system wherein a portion of the digital content data is substituted with token data to generate tokenized data. The tokenized data are stored at predetermined memory locations on the system to deter unauthorized access of the digital content data.

The present invention is further directed to a method and system for preventing unauthorized use of digital content data hosted on a system wherein an operating system interface operating on the system and the digital content data at an assassin process are monitored to determine whether an unauthorized attempt at accessing the digital content data occurs. In the event of unauthorized access, the unauthorized access is deterred and communicated to the operating system interface.

The present invention is further directed to a method and system for preventing unauthorized use of digital content data in a system having memory locations wherein the system is scanned to determine available memory locations based on a file system identifying locations of files on the system. Target memory locations are determined within the available memory locations at which to store the digital content data. The digital content data is stored at the target memory locations.

In another aspect, the present invention includes a software development kit and toolkit, which embodies the aspects of the inventions described above and allows for their application to target content without revealing the details of the construct methods to the user.

The present invention is thus further directed to a system for preventing unauthorized use of digital content data in a system having memory locations wherein the system enables a user to select from a plurality of tool modules, each module providing a service for protecting digital

5 content from unauthorized use such that a user can protect digital content. The tool modules may comprise modules that perform functions selected from the group of functions consisting of: interleaving; tokenization; obfuscation; saturation; translocation; shimming and assassination.

The present invention is further directed to systems and methods that allow for the

10 delivery of content in a fashion that prohibits content modification and duplication by unauthorized persons. The invention mechanisms detailed in this document enable, support and secure the delivery of software titles, audio, video, and text/graphic/e-book/e-presentation formats using both hard media and network content delivery models.

The present invention further processes and packages the components of a digital content

15 product, for example the standard component contents of a hard media digital product, including executable files, documentation files, image files, and audio files. A standard hard media product may be taken in entirety from a CD release and converted into a securely downloadable product. Some or all of the content is indelibly watermarked with serialized purchase information unique

20 to each purchaser at the download site before being downloaded. On a server that deploys this protected content, kit components can be packaged as large archives or can be stored individually (in the same form as a hard media kit, including optionally, directory structures) and then manufactured on-demand, per-user, per purchase. The final kit is packaged as a collection of encrypted archives, or as a single monolithic archive, securely encrypted, and made installable

25 at the appropriate time by a secure installation process. Each installation of the product can optionally be made more secure by requiring authentication; multiple invention methods may be used including network authentication and authentication from locally hidden data and/or local computing device and peripheral configuration information. In the network installation case, installation or re-installation may be disallowed at any time by the vendor based on their criteria

(number of times, frequency, etc). Such remote authentication invention methods may be added to hard media based products as well.

5   The present invention further allows for modification of the product's files, both before the download while still on the server (or before being copied to the server), and also the application files in the product directory after the installation, on the customer computer. This invention inserts hidden data into these product files, this hidden data incorporating among other identifying data a securely encrypted transaction ID, which may also be modified by a function based on information about the target system's component-specific configuration information.

10   The hidden data may alternately or inclusively be a simple numeric index or may also have meaningful content interleaved into itself, such data hiding concepts defined herein. The data may be of any length. These hidden data items are inserted into secret locations within the product files prior to kitting, for example using the mechanisms disclosed herein, at the point of purchase.

15

The present invention further authorizes the execution of product components by providing a service that correlates attributes of the executable product with the system which is the target of the execution of the product. This applies to hard media and to Electronic Software Distribution (ESD). In this aspect, the ESD content delivery phase may be authenticated by

20   means of purchase (or other) unique transaction-specific information and/or system specific information.

The present invention further processes product files in order to make room for more hidden data items. These additional reserved spaces for hidden data items are integrated directly

25   into any desired product files and are optionally pre-filled with filler content

The present invention further segments the contents of the download kit such that specific small and critical files or even portions of such files are segregated from the main kit. The downloaded installation kit is therefore incomplete in small but critical ways. The installation

11

process requires subsequent authenticated reconnections to the download host, followed by small and volatile downloads of these critical items.

The present invention further segments the contents of the installed digital product such that specific critical files and/or portions of such files are segregated and encrypted in a fashion that makes the installed product only function properly on the intended target system.

According to the present invention, certain chosen program elements are intentionally incomplete, and will be completed by means of executable information extracted from the authorization process, in some cases by hiding the information within the authentication response. For example the authorization process can provide the system with both numerical encrypted information (keys for further system decryption use) and executable content critical to task completion.

According to the present invention, content has certain sections altered such that key elements are removed and hidden elsewhere, (on the media itself in the case of hard media, on the network in other cases, on other system storage devices in the case of something already installed on a computer system) in secret locations. Execution requires these hidden elements be found and replaced in their original locations within the content. These elements are stored in locations that would not be copied easily with either the installer media or the installed product directory.

The present invention is further directed to mechanisms that detect the presence of classes and instances of software development tools (known variously as ICES, Debuggers, dump/lift tools, process fixup tools) and which initiates responses (exit, kill intrusion process, etc) when invoked on a system that has been thus instrumented for hacker purposes.

The present invention further determines whether the environment is safe (criteria include absence of some or all software development tools and emulation environments) and allows the protected title to run. After this occurs, subsequent execution of any and all prohibited tools is

disallowed in part this is accomplished by means of methods discussed in the Translocation claims attached herein; by translocation of the desired tool with a stub that exits. Other methods include disabling certain input device (keyboard and mouse) responses as needed.

5   In another aspect, in order to defend the system from attack, the system exits upon being compromised or otherwise touched by unauthorized tools or methods. The exit itself may be delayed or deferred to obfuscate the logic behind the exit process. Other cooperating components of the invention (processes, threads, tasks and other logical algorithmic entities) can be configured such that if one exits for any reasons all the others exit as well.

10

In another aspect, all system defense related tasks (such as encryption, decryption, message passing, debugger detection via memory scan, etc) are encapsulated within other routines commonly used by the system. For example, it can be that every file open also triggers a defensive routine that scans memory or rewrites memory. In this manner, any and all system

15   activity act as events that trigger defensive routines, so these routines do not necessarily have to poll or loop as their exclusive method of acting upon the system. Removal of these defenses is non-trivial as they can be deeply integrated into every aspect of the system.

In another aspect, each process, thread or task in the system has a dual or multiple role. One is the true functional role of that component (such as decryption), and the other is the

20   monitoring and protection of all other parts of the system using techniques outlined in this document. Such protective functions are sometimes referred to as Assassin processes. Any attempt to compromise the system will result in a mass exit of all system components. The distributed nature of this protection across dozens of system tasks results in a very powerful

25   redundant protection model where any attempt to tamper with one part of the system results in a protective response from the rest of the system.

In another aspect, all strings and other resource elements used are encrypted and decrypted by the system in a volatile fashion when used, and then disposed of, such that they

30   cannot be easily searched for within the code either statically or in memory.

13

In another aspect, data values that are critical to the system are read and rewritten by a number of decoy or spoof processes, such that debugger watchpoints on these values, if any, will be triggered excessively, and thus it will be difficult to determine which accesses are decoy and which are valid without much deeper debugging.

In another aspect, system and product code can maintain itself in a difficult-to-modify state even if modification is attempted by a sophisticated debugger, editor or other tool. Key code elements are rewritten in place, in memory, using whatever mode of privilege is required, many times per second (tens, hundreds, tuned to be optimal as needed), at initialization and during execution, so that any attempts to tamper the code will be changed back to the original state. Depending on the nature of the change, the system may also choose to exit as a result of the tampering. For example, a classic hacker attack, the modification of Import Tables, is defeated in this way. All key code segments are duplicated in an encrypted archive, the archive is hidden (perhaps within files, between files, or outside the file system) read from that archive (some part of the read and decryption occurs in the virtual machine context described elsewhere in the document). Decoy archives and decoy read processes are also established which read from nonencrypted decoy code and write it over the sections, or seem to write it over the sections (writes through the I/O subsystem which are then defeated by tapping into the subsystem and tossing the data away) such that attempts to modify these decoy archives result in no change to the running code.

In another aspect, certain critical executable components are processed before shipment to be populated with tens or hundreds of thousands of data values which trigger debugger breakpoints in many debuggers. During normal execution of the title in a non-debug environment, these breakpoints are handled by a null handler and little negative performance impact is achieved. In the debug environment, each breakpoint stops the debugger and requires the intruder to at the least click the mouse and type into the keyboard. A single execution of such a title would require on the order of a hundred thousand mouse-clicks and keyboard presses. The

14

purpose of such is to significantly deter unauthorized debugging, and at the very least to render it as slow and painful as possible.

In another aspect, resistance to tools used to "memory lift" or "memory dump" is achieved by modifying (corrupting) large parts of the code before packaging and storing the original correct parts elsewhere. This modification can take the form of gross and/or subtle corruption, yielding unexecutable code or subtle logical alterations in code that runs. When the code is run in the correct context, a cooperating synchronized system process modifies the code back to the correct executable state but only in a rolling window of context such that at no time is the entire body of the content correct, just those parts that are required at the current execution time. Once executed these lines of code are re-corrupted.

In another aspect, source, object, or executable code is processed to generate variant different versions of executable code, by means of replacement of content with functionally synonymous content. For example in the case of executable content, different assembly language instructions and ordering, that produce the same functional outcome are generated, such that no two such versions share the same fingerprint or the same code-line-number relationship per instruction. This variation is designed to reduce or eliminate the broadly disseminated effectiveness of hacker tutorials and documents that usually depend on specific line-number directions.

Brief Description of the Drawings

The foregoing and other objects, features and advantages of the invention will be apparent from the more particular description of preferred embodiments of the invention, as illustrated in the accompanying drawings in which like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention

FIG. 1 is a block diagram of a computer system or consumer computerized appliance device to provide an understanding of how the systems and methods of the invention interact with such devices.

FIG. 2 is a diagram demonstrating the flow of digital content from its delivery media through a computer system such as the one in FIG. 1, in accordance with the present invention.

FIG. 3 is a flow diagram that describes the creation of an interleaved, multiplexed, encrypted content stream such as those used for information hiding and content watermarking, in accordance with the present invention.

FIG. 4 is a block diagram illustrating the placement of hidden, stored content, in accordance with the present invention.

FIG. 5 is a block diagram illustrating an alternative or additional placement method for hidden, stored content, in accordance with the present invention.

FIG. 6 is a flow diagram illustrating the storage of digital content in a hidden, secure manner, in accordance with the present invention.

FIG. 7 is a flow diagram illustrating a method for retrieving such hidden, stored content, in accordance with the present invention.

FIG. 8 is a block diagram illustrating four related methods of securing an encrypted watermark or encrypted stream, in accordance with the present invention.

FIG. 9 is a block diagram illustrating three related methods for translocating content in a secure fashion, in accordance with the present invention.

FIG. 10 is a flow diagram that illustrates a method to prepare content for translocation, in accordance with the present invention.

FIG. 11 is a flow diagram illustrating a method to invoke and utilize translocated content, in accordance with the present invention.

FIG. 12 is a flow diagram illustrating a method to tokenize and obfuscate content, in accordance with the present invention.

FIG. 13 is a detailed flow diagram illustrating a method to tokenize and obfuscate content, in accordance with the present invention.

FIG. 14 is a further detailed flow diagram illustrating a method to tokenize and obfuscate content, in accordance with the present invention.

FIG. 15 is a high level flow diagram illustrating a method to utilize previously tokenized and obfuscated content, in accordance with the present invention.

FIG. 16 is a detailed flow diagram illustrating a method to utilize previously tokenized and obfuscated content, in accordance with the present invention.

FIG. 17 is a flow diagram illustrating a method to saturate logging and debugging tools and techniques as a method of providing additional security, in accordance with the present invention.

FIG. 18 is a detailed flow diagram describing a method to saturate logging and debugging tools and techniques as a method of providing additional security, in accordance with the present invention.

FIG. 19 is a further detailed flow diagram describing a method to saturate logging and debugging tools and techniques as a method of providing additional security, in accordance with the present invention.

FIG. 20 is a detailed control flow diagram describing a method to saturate logging and debugging tools and techniques as a method of providing additional security, in accordance with the present invention.

FIG. 21 is a flow diagram describing the aspects of this invention that allow for the secure attachment (hooking) of device shims, operating system shims, and device driver shims, in accordance with the present invention.

FIG. 22 is a flow diagram describing the aspects of this invention that allow for the security obfuscation of the activity of device shims, operating system shims, and device driver shims.

FIG. 23 is a flow diagram describing a mechanism used to prevent the execution of, or access to, content that is disallowed, or to redirect access to other content in a fashion transparent to the accessing party or process, in accordance with the present invention.

FIG. 24 is a flow diagram that illustrates a method for the creation of protective "assassin" processes, in accordance with the present invention.

FIG. 25 is a flow diagram that describes methods that determine authorization for access to content, in accordance with the present invention.

FIG. 26 is a flow diagram that describes methods that determine authorization for access to content, in accordance with the present invention.

FIG. 27 is a flow diagram of a method that takes as input the standard contents of a digital hard media product (including but not limited to software, e-books, entertainment and game media, etc) and produces as output a securely downloadable digital content product, in accordance with the present invention.

FIG. 28 is a flow diagram of a method that establishes the unique identity of the Target Computing device, in accordance with the present invention.

FIG. 29 is a flow diagram of a method that processes digital content components as part of the kitting process for electronic content distribution, in accordance with the present invention.

FIG. 30 is a flow diagram of a method that assigns unique identifying values to individual subcomponents of the Target Computing device, in accordance with the present invention.

FIG. 31 is a flow diagram of a method that inserts unique identifying data into digital content product components, in accordance with the present invention.

FIG. 32 is a flow diagram of a method that authenticates access to and use of digital content by verifying unique identifying data found within the digital content, in accordance with the present invention.

FIG. 33 is a flow diagram of a method that provides authentication data to the method of FIG. 32, in accordance with the present invention.

FIG. 34 is a flow diagram of a method that creates additional space within digital content for the later insertion of unique identifying data, in accordance with the present invention.

FIG. 35 is a flow diagram of a method that inserts unique identifying data into digital content on a server, using such created locations as those created in the flow diagram of FIG. 34 or other space as found within the content, in accordance with the present invention.

FIG. 36 is a flow diagram of a method that inserts unique identifying data into digital content as it is being installed onto a Target Computing device, using such created locations as those created in FIG. 34 or other space as found within the content, in accordance with the present invention.

FIG. 37 is a flow diagram of a method that renders incomplete and thereby prohibits execution of or use of digital content on systems other than those authorized to execute it, in accordance with the present invention.

FIG. 38 is a flow diagram of a method that encrypts and hides unique system and acquisition transactional identifying information, in accordance with the present invention.

FIG. 39 is a flow diagram of a method that scans memory on a Target Computing device to determine whether certain prohibited executable applications, tools, and files are present, in accordance with the present invention.

FIG. 40 is a flow diagram of a method that determines whether the system is an actual or virtual computing device, in accordance with the present invention.

FIG. 41 is a flow diagram of a method that disables the use of certain keystroke sequences, in accordance with the present invention.

FIG. 42 is a flow diagram of a method that disables the keyboard entirely when desired for certain input window or dialogue focus configurations, and selectively allows it for others, in accordance with the present invention.

FIG. 43 is a flow diagram of a method that disables mouse button function entirely when desired for certain input window or dialogue focus configurations, and selectively allows it for others, in accordance with the present invention.

FIG. 44 is a flow diagram of a method that detects compromise of the system in the form of the exit of other system components and which itself then initiates a cascading exit event, in accordance with the present invention.

FIG. 45 is a flow diagram of a method that allows cooperating system component to more securely write data to one another, in accordance with the present invention.

FIG. 46 is a flow diagram of a method that allows cooperating system component to more securely read data from one another, in accordance with the present invention.

FIG. 47 is a flow diagram of a method that allows cooperating system component to more securely write data to one another, using further levels of indirection than those shown in FIG. 45, in accordance with the present invention.

FIG. 48 is a flow diagram of a method that embodies security or security system functions within any standard system function, in accordance with the present invention.

FIG. 49 is a flow diagram of a method that embodies exit functions as in FIG. 44 within any standard system function, in accordance with the present invention.

FIG. 50 is a flow diagram of a method that converts system resources such as strings into encrypted resources to reduce their search vulnerability and comprehension, in accordance with the present invention.

FIG. 51 is a flow diagram of a method that renders encrypted resources such as those created in FIG. 50 usable as needed, in accordance with the present invention.

FIG. 52 is a flow diagram of a method that touches many memory locations in order to generate excessive debugger event traffic, in accordance with the present invention.

FIG. 53 is a flow diagram of a method that overwrites data in memory with such rapidity and frequency that attempts to alter this in memory data via unauthorized means are eradicated automatically, in accordance with the present invention.

FIG. 54 is a flow diagram of a method that inserts a number of breakpoints into target digital content in order to render unauthorized debugging extremely difficult, in accordance with the present invention.

FIG. 55 is a flow diagram of a method that protects digital content from being memory lifted by creating a "rolling window of corrected code" in an otherwise corrupted body of digital content, in accordance with the present invention.

FIG. 56 is a flow diagram of a method that creates multiply variant digital content, in order to increase the difficulty of cooperative debugging and cracking of digital content when deployed, in accordance with the present invention.

Detailed Description of Preferred Embodiments

The present invention will be more completely understood by means of the following detailed description, which should be read in conjunction with the attached drawings, FIG.1 through FIG. 56, in which similar reference numbers indicate similar structures.

This invention and its embodiments may be implemented on a personal computer or general purpose digital computer as shown in **FIG. 1**, including, but not limited to, single- or multiple-processor-based Windows, Linux or Macintosh desktop computers such as those found with increasing frequency in contemporary homes and offices. Embodiments of this invention may also be implemented on a digital processing circuit, including, but not limited to, those found in CD and DVD consumer audio/video appliance components or systems, stationary or mobile applications. Embodiments of this invention are also well suited for implementation on

20

other computing appliance devices such as hard-disk or random access memory based video and audio entertainment appliances which may be digital-processing-circuit based, or may be based on general-purpose digital computing architectures. As can be made clear to one skilled in the art, this invention is applicable to all digital content uses, because all such uses have the same

5   basic elements; the content 7 is input to the system in some fashion as shown in **FIG. 2,** stored for some period of time in the system's memory **8** (whether disk, volatile RAM of any kind, or non-volatile RAM of any kind), and executed on a processor **9,** whether the main processor of the system, or an auxiliary processor, and whether the content itself is directly executable on the processor or is executed within a helper application (such as an audio, video, or word processing

10   application, depending on content type).

The systems and methods of the present invention may be embodied and implemented on a general-purpose digital computer or personal computer system **6** as shown in **FIG.1.** Such a system commonly includes an input device **1** (one or more may be connected; this includes anything which provides external content and data to the computer as input, such as a mouse or

15   keyboard or scanner). Such a computer system 6 also has as a subcomponent a collection of software and hardware components **5** that comprise the processor, all system bus and cache lines, and the running operating system and all of its subcomponents. Output is presented to the user via one or more output devices **4,** which include, but are not limited to, the computer's display

20   (CRT or LCD) and the hardware that drives it, and can also include printers, speakers and sound cards, and radio frequency, S-video, component, or digital video outputs for consumer/entertainment applications and devices.

The computer system **6** may be a general purpose home or office or mobile computer

25   system. Such systems allow for the usage/consumption/execution of a variety of forms of digital content; the invention disclosed herein can be applied to all forms of such digital content and the foregoing will describe some of the forms of this content on this computing platform family. Such systems are generally multiple-component level hardware-based systems, comprised of a motherboard or main-board, with various specialized components (such as I/O cards, video

30   cards, processors, memory) attached to it by means of connectors. Each such card and the

motherboard itself and the attached components have some amount of executable firmware located on various non-volatile memory 3 integrated circuit components, but the majority of the system's operational logic is driven by executable operating system code that is stored on media (non-removable or removable magnetic and or optical media, or non-volatile random access

5   memory media). Usually on a system of this general type such executable code is created by software developers and is written using program code in modern programming languages such as C and C++. Such languages are programmatically compiled into assembly language or machine instruction code and are later executed directly on the system's central processing unit. Other programming languages and techniques, such as those used in Java, JavaScript, and Visual

10   Basic, are interpreted at runtime; they're stored in their original language, or in a moderately tokenized version of their original language, and are then rendered on the fly at execution time into assembly language or machine instruction code and are later executed directly on the system's central processing unit. Other forms of relevant digital content utilized on such a computer system are audio (for example .wav or .mp3 file formats), video (for example .avi file

15   format), e-book and documentation (for example .pdf or variant secure-portable-document-format), and all such content may be significantly security-enhanced by the application of the invention described in this document.

       As shown in **FIG. 2**, a computing system **10** of any kind, whether a general purpose

20   computer **6** (see FIG. 1) or an appliance device with computing capability and components (such as a DVD or CD player) is commonly used to consume, execute, display or otherwise utilize digital content. Digital content **7** (including but not limited to the above examples) is made available to the system by a variety of means including by network transmission (internet or intranet), on hard media, on non-volatile random access memory removable storage (such as the

25   compact flash standard for removable media storage cards); and is read from that media **7** into the system's memory **8**. In the case of such content which is unprotected, the utilization model is straightforward; it is read from the input media **7** into memory **8** and then executed at some point thereafter. This document will define the word "executed" to mean, in the case of binary executable program content (for example a computer video game, or a game console video game

30   running on a game console computing appliance device, or a word processing program intended

to run on a general purpose computing device), executed on the processor **2** as a program; in the case of readable document formats (for example a Word .doc file or an Acrobat .pdf file) executed within the appropriate application, which in turn executes on the processor **2** as a program; in the case of all other digital content types (for example audio, video) they too are

5 intended to be input to an appropriate application (for example on a general purpose computing device, a software application such as Windows Media Player; in the case of a computing appliance device such as a DVD player or a game console, a firmware executable which runs on a processor **2** within the computing appliance device) which in turn executes on a processor **2** within the computing platform. Also note that within this document the term "stream" may be

10 used interchangeably with the term "file" to represent a collection of bits that represent some form of digital content, including not limited to standard file types found on operating systems such as Windows and archive or container formats used to convey content on the internet such as "ZIP" files or "TAR" files.

15 In one embodiment of this invention, illustrated in **FIG. 3**, an interleaved-multiplexed data hiding process **19** (optionally, also, an excellent framework for the application of encryption to the interleaved, multiplexed content) is provided that performs multiple functions detailed in the foregoing paragraphs. The system and process of the present invention create meaningful (optionally encrypted) data-identifier tags, sometimes referred to as watermarks, for later

20 insertion into content, of any desired size in number of bytes, each of which have an individual variation even when the identifier data is identical for each. Data content is first input as shown in step 11. Watermarks are defined as composed of a variable number of bits **12**. These collections of bits are re-ordered as needed and interleaved at step **13** with other data, that is either randomly generated, or time-stamped, to create a unique numeric value. Alternatively, the

25 collections of bits can be interleaved at step 13 with data streamed directly from other portions input data content **11** itself, to be hidden in the watermark. A simple verification value is incorporated into the watermark data or the interleaved-multiplexed data stream such that any instance of a watermark may be examined to determine if it has been tampered with. Following this, the resultant stream is output and written to predetermined memory locations at step 18

either at locations as selected in the mapping process outlined elsewhere in this document or any other locations specified by the system.

Prior to writing the output stream, the watermark may optionally be encrypted by a key to further enhance its security. The encryption key itself can also be optionally encrypted in a similar manner in steps 15 (subdivide into segments) 16 (interleave) and 17 (encrypt), and optionally stored in a known location with the data stream **18**.

An example of the resultant effect of the system and method of the invention is provided in the following illustration. Assume an identifier 1234 **11** that is to be hidden in 100 locations on a game CD (see description below in connection with **FIG. 6, FIG. 7, FIG. 8** for details related to where and how the invention elects to hide such data). Assume also a subdivision size of 8 bits, and a total number of streams to be interleaved at 2 streams. The example of this method takes the bytes of the identifier, in this case the bytes "1", "2", "3", and "4" **12** and interleaves them with a second stream of bytes **13**. These four divided subcomponents are then interleaved **13** with some other data; in this example the data comes from the text of this sentence beginning with "These four divided" **11**. Thus the first watermark generated would be "T1h2e3s4" **13** and the second watermark would be "e1 2f3o4" **13**. Even in this simple form it is clear that the two watermarks have a different appearance and would not be trivially searchable; however when optionally encrypted at step **14** they become utterly dissimilar, yielding the values "aJt6G2.R" and ">*qI1Ub$" in this example; these two values, hidden (see **FIG. 6**) or stored in the file system (see **FIG. 4**) would be quite secure, yet each is easily locatable by means of this invention (the location process is described with reference to FIG. 7, below), and once located, each is easily translatable using the invention components described with reference to **FIG. 7** back into the identifier "1234".

The present invention, illustrated in **FIG. 3**, also serves as a means of interleaving N streams of data for purposes far more general, and more broadly useful, than simply watermarking content. It can irrevocably intermix **13** multiple streams **11** of content such that

24

they remain interleaved until utilized by an appropriate component of the present invention, as illustrated in **FIG. 7**, below.

The following code example details an embodiment of this invention which illustrates the
5   concepts discussed in the above paragraphs which reference **FIG. 3.** This embodiment is tuned to subdivide a stream of data into 8 bit bytes and then interleave them; in practice, any number of streams may be subdivided, and any subdivision value may be used.

```
    // Return a sig
10    BOOLEAN CSigGen::GetSig(
        const BYTE*const    inp_bld,        // sig data
        const unsigned int  in_cbld,        // length of sig data
        BYTE*const          outp_bSig,      // generated sig, SigSize() bytes
        const DWORD                      in_dateTime,    //      The date time bytes
15    const int                        in_sigToggle    //      Double the size of a watermark
    )
    {
        BYTE        abJumble[MAX_SIG_SIZE]; //  buf for jumble dat
        BYTE        abSigRaw[MAX_SIG_SIZE]; //  buf for in-process sig
20    BOOLEAN     bStat;
        unsigned int  cbJumb;
        unsigned int  cbSig = SigSize();    //  size of gen'd sig
        unsigned int  ii;
        unsigned int  iTotal;
25    unsigned int  jj;
        unsigned int  cbld = min(SigSize()/2, in_cbld);


        // Validate args
30    if ( (NULL == outp_bSig)   ||
            (cbld > cbSig)      ||
            (MAX_SIG_SIZE < cbSig) ||
                ((in_sigToggle == 1) && (in_cbld < 2*cbld)))
        {
35        return FALSE;
        }

        // Get the jumble data we need
        cbJumb = (cbSig - cbld) - 1;    // subtract 1 for checksum
40    if (!m_pJumbler->GetData(cbJumb, abJumble))
        {
            return FALSE;
        }

45    // Compute the simple verification value of the data
        iTotal = 0;
        for (ii = 0; ii < cbld; ii++)
```

```
    {
        iTotal += (unsigned int)(inp_bld[ii + in_sigToggle*cbld]);
    }
    abJumble[cbJumb] = (BYTE)((unsigned int)0x00FF & iTotal);

    // Interleave if the sizes are right
    if ( cbld == cbSig / 2)
    {
        for (ii = 0; ii < in_cbld; ii++)
        {
            jj = 2 * ii;
            abSigRaw[jj   ] = inp_bld[ii + in_sigToggle*cbld];
            abSigRaw[jj + 1] = abJumble[ii];
        }

                if ((in_dateTime) && (cbSig >= 16) && (in_sigToggle == 0)){
                        // Instead of using random data, use the date/time bytes
                        abSigRaw[1] = (BYTE) (in_dateTime & 0xff);
                        abSigRaw[5] = (BYTE) ((in_dateTime & 0xff00) >> 8);
                        abSigRaw[9] = (BYTE) ((in_dateTime & 0xff0000) >> 16);
                        abSigRaw[13] = (BYTE) ((in_dateTime & 0xff000000) >> 24);
                }
                else if ((cbSig >= 16) && (in_sigToggle == 1) && (in_cbld == cbld*2 + 4)){
                        // Instead of using random data, use the date/time bytes
                        abSigRaw[1] = inp_bld[16];
                        abSigRaw[5] = inp_bld[17];
                        abSigRaw[9] = inp_bld[18];
                        abSigRaw[13] = inp_bld[19];
                }
    }

    // Otherwise, tack the jumble data on the end
    else
    {
        memcpy(abSigRaw, inp_bld, cbld);
        memcpy(&(abSigRaw[cbld]), abJumble, cbSig - cbld);
    }

    // Now encrypt it
    bStat = m_pEncryptor->EncryptBlock(abSigRaw, outp_bSig);

    // Zero the in-process sig data
    memset(abSigRaw, 0, sizeof(abSigRaw));

    // Done
    return bStat;

} // End GetSig()
```

26

A simple example and embodiment of this aspect of the present invention now follows. Assume three streams of digital content, in this case three files on disk, each of five megabytes in size. File "A" is a text file. File "B" is an audio file. File "C" is a Word document; thus on a general purpose computing device 6 (see FIG. 1) Windows operating system this yields the three

5   hypothetical input streams 11 derived from A.txt, B.wav, C.doc. Each such stream is subdivided into segments of M bits in length 12, and interleaved as in the previous example. The resultant output, even prior to encryption, is clearly incomprehensible to any mechanism other than this invention (see, for example, the operation disclosed in **FIG. 7**) due to the nature of the mixed text, audio, and document data. Even so, the output itself may be encrypted as in **FIG. 3**, steps

10   14, 15, 16 to further protect its contents.   The aggregate stream is optionally encrypted, and then the keys necessary to decrypt this stream, if encrypted, are themselves encrypted and hidden; the manner of the hiding process may be as described in FIG. 8, examples 42, 43, 44 or 45, described in detail below, or the key may be hidden in another location known to the system as needed. This aggregate multiplexed stream, now fifteen megabytes in size may be written 18 at

15   this time.

One embodiment of the writing process 18 streams the contents back into the original files A, B and C (see FIG. 6 and corresponding description) from where they came, without regard for which contents came from which files, such that the first five megabytes of the fifteen

20   megabyte stream is used to fill A.txt, the second five megabytes is used to fill B.wav, and the third five megabytes is used to fill C.doc. The method used to determine where to write, to keep track of where the data was written, and to record the manner in which it was interleaved, is detailed below with reference to **FIG. 6**. After having written the content, the present invention supports multiple techniques for providing that the data may be later read and de-interleaved

25   properly (see **FIG. 7**, below). Note that the concept of a map of locations and interleaved data information as detailed in **FIG. 7 40** is optional for purposes of this aspect of the present invention. The map can be incorporated into the stored, hidden content, or as an alternative embodiment of the invention, algorithmic logic identical to that described below in **FIG. 6**, with the order of execution as in steps **27, 28** (described below) is incorporated into the process of the

30   present invention such that the likely map locations can be determined based on the context and

content of the media. The retrieval of segments of the stream can then be attempted the simple verification values calculated as shown in the code example above to determine that the correct data has been retrieved. The stream contents can be retrieved, decrypted, de-interleaved, and utilized.

5

The following example CmapLocation::WriteFile is a code example of the logic used to create such a map file of locations. Note that there are two types of maps created by the CmapLocation::WriteFile code example below: raw maps and location maps. Raw maps are built upon a linked list structure of locations and lengths and also contain detailed information

10   about the file this mapped area was derived from. Location maps are a further abstraction, and are built upon linked lists of raw map lists, where each location map entry contains information to locate a certain number of data bytes. In the example code below, this value is 16 bytes to support the example encryption method, which is optimized for 16 bit units of data. So in the foregoing example, the location map is created from the raw map by partitioning it into 16 byte

15   blocks. These 16 byte blocks need not be contiguous.

Also note that the following code examples embody another aspect of this invention; namely, a file locker, a mechanism as described below with reference to **FIG. 8** and touched upon in **FIG. 3** steps **15, 16, 17**. The file locker serves to securely marry the decryption key to

20   an encrypted stream such that the process described in **FIG. 7** can successfully unlock the data and decrypt it. The file locker further encrypts the encryption key using a secondary encryption algorithm, with a known key, and hides the key information within the encrypted stream as described below with reference to **FIG. 8**. The encrypted key may be hidden whole (as in steps 42, 43, and 44 of FIG. 8) or may be further subdivided and hidden in a scattered fashion (as in

25   steps 45, 46, 47, 48, 49, and 50 of FIG 8).

```
CMapLocation::WriteFile(
    const char*const   mapFileName
)
{
    LocationMapList *  pos = locationMapList;
    MapRawList_t *     rpos;
    BYTE output[512];
```

28

```
        CFileLock *fileLocker;
        C2Encryptor *fileEncrypt;
        CREncryptor *fileLock;
        BYTE key[16];
5       int i;
        unsigned long j;
        WORD        majorVersion = HIWORD(MAP_LOC_VERSION);
        WORD        minorVersion = LOWORD(MAP_LOC_VERSION);

10      // Encryption Locker
        fileLock = new CREncryptor(MAP_LOC_KEY);
        // Generate Random key
        srand( (unsigned)time( NULL ) );
        for (i=0;i<16;i++) {
15          key[i] = (char) (rand() / (RAND_MAX / 255));
        }

        fileEncrypt = new C2Encryptor(key, 16);

20      if (mapFileName)
        {
            fileLocker = new CFileLock(fileEncrypt, key, 16, fileLock, majorVersion, minorVersion, (char *)
        mapFileName);
        }
25      else
        {
            fileLocker = new CFileLock(fileEncrypt, key, 16, fileLock, majorVersion, minorVersion,
                "c:\\l.tmp");
        }
30
        // Write out location size
        fileLocker->WriteBytes((BYTE *) &(locationSize), sizeof(locationSize));

        while (pos && pos->locNumber)
35      {
          if ((pos->location->length == locationSize) && (pos->link) &&
              (pos->link->location) && (pos->link->location->length == locationSize) &&
              ((pos->location->offset + pos->location->length) == pos->link->location->offset))
          {
40          // Run of location map entrys
            output[0] = _MARKER;
            output[1] = LOCMAPRUN;
            fileLocker->WriteBytes(output,2);
            fileLocker->WriteBytes((BYTE *) &(pos->location->offset), sizeof(pos->location->offset));
45
            j = 2;
            pos = pos->link;
            while ((pos->location) && (pos->location->length == locationSize) && (pos->link) &&
                (pos->link->location) && (pos->link->location->length == locationSize) &&
50              ((pos->location->offset + pos->location->length) == pos->link->location->offset))
            {
              j++;
              pos = pos->link;
            }
```

29

```
            pos = pos->link;
            // Write out number of entries in this run
            fileLocker->WriteBytes((BYTE *) &(j), sizeof(j));

        }
        else
        {
            // Normal location map entry
            output[0] = _MARKER;
            output[1] = LOCMAPENTRY;
            fileLocker->WriteBytes(output,2);
            fileLocker->WriteBytes((BYTE *) &(pos->locNumber), sizeof(pos->locNumber));
            rpos = pos->location;
            while (rpos) {
                if (rpos->length > 0)
                {
                    output[0] = _MARKER;
                    output[1] = LOCMAPLOC;
                    fileLocker->WriteBytes(output,2);
                    fileLocker->WriteBytes((BYTE *) &(rpos->offset), sizeof(rpos->offset));
                    fileLocker->WriteBytes((BYTE *) &(rpos->length), sizeof(rpos->length));
                }
                rpos = rpos->link;
            }
            pos = pos->link;
        }
    }
    output[0] = 0;
    fileLocker->WriteBytes(output, 1);  // Write a null byte out at the end of the file
                                        // to cause read back of file to end

    delete fileLocker;
    delete fileEncrypt;
    delete fileLock;
}


CMapRaw::WriteFile(
    const char*const    mapFileName
)
{
        MapRawList_t *pos = m_rawMapList;
        BYTE output[512];
        CFileLock *fileLocker;
        C2Encryptor *fileEncrypt;
        CREncryptor *fileLock;
        BYTE key[16];
        WORD stringLength;
        int i;
        WORD                majorVersion = HIWORD(MAP_RAW_VERSION);
        WORD                minorVersion = LOWORD(MAP_RAW_VERSION);

        // Locker
        fileLock = new CrEncryptor(MAP_RAW_KEY);
```

30

```
      // Generate Random key
      srand( (unsigned)time( NULL ) );
      for (i=0;i<16;i++) {
            key[i] = (char) (rand() / (RAND_MAX / 255));
5     }

      fileEncrypt = new C2Encryptor(key, 16);

      if (mapFileName)
10    {
            fileLocker = new CFileLock(fileEncrypt, key, 16, fileLock, majorVersion,
            minorVersion, (char *) mapFileName);
      }
      else
15    {
            fileLocker = new CFileLock(fileEncrypt, key, 16, fileLock, majorVersion,
                  minorVersion, "c:\\r.tmp");
      }

20    while (pos)
      {
            if (pos->length > 0)
            {
                  if (pos->name)
25            {
                        output[0] = _MARKER;
                        output[1] = FILENAMETAG;
                        fileLocker->WriteBytes(output,2);
                        stringLength = strlen(pos->name);
30                      fileLocker->WriteBytes((BYTE *) &stringLength, sizeof(WORD));
                        fileLocker->WriteBytes((BYTE *) pos->name, stringLength);
                  }
                  if (pos->fileStartAddress) {
                        output[0] = _MARKER;
35                      output[1] = FILEINFOTAG;
                        fileLocker->WriteBytes(output,2);
                        fileLocker->WriteBytes((BYTE *) &(pos->fileStartAddress), sizeof(pos-
>fileStartAddress));
                        fileLocker->WriteBytes((BYTE *) &(pos->fileLength), sizeof(pos->fileLength));
40                }
                  output[0] = _MARKER;
                  output[1] = RAWMAPENTRY;
                  fileLocker->WriteBytes(output,2);
                  fileLocker->WriteBytes((BYTE *) &(pos->offset), sizeof(pos->offset));
45                fileLocker->WriteBytes((BYTE *) &(pos->length), sizeof(pos->length));
                  output[0] = pos->flags;
                  fileLocker->WriteBytes(output, 1);
            }
            pos = pos->link;
50    }

      delete fileLocker;
      delete fileEncrypt;
      delete fileLock;
```

31

```
//fclose(m_rawFile);
}
```

5   With reference to **FIG. 4** , the present invention includes a system and method by which

content can be hidden or stored in a variety of locations, both intrafile (within a file) and interfile

(between files) and also outside the file system on devices that support extra-files system access

(such as ISO-9660 CD discs). The map files in the code example above detail how such

locations are represented and communicated.

10

        The operation for choosing the actual locations will now be described with reference to

FIG. 5. Note that in **FIG. 5** the extra-file system locations **26, 25** are excellent locations to store

content securely, because application programs generally cannot access the raw data and are

limited to accessing only those data items that are located within the bounds of the file system **24**

15   as known to the table of contents **23**. All application file system accesses through normal

interfaces, for example the Windows application interfaces to Read(), Open(), and Close() a file,

require a file handle or descriptor, which means that most applications can only access areas of

the file system known to the table of contents **FIG. 5 23**. Thus, on any supported file system

format, for example ISO-9660, liberal use is made of any extra-file system space that may be

20   available.

        With reference to FIG. 6, an aspect of the present invention is disclosed that is used to

hide or store information in secure or non-obvious locations. In a first step of this aspect, the file

system is scanned all the possible locations appropriate for information hiding are determined 27.

25   Desired locations from among all the possible locations **28** are selected the ordering of insertion

into these locations **28** is determined. The stream of interleaved data, described above with

reference to FIG. 3, may optionally be encrypted as desired **29**. Next, low-level operating

system interfaces are accessed and device level access **30** is initialized at a level far below the

normal file system interfaces, such that the device may optionally be addressed in any and all

30   valid raw physical locations, whether inside or outside the standard file system. In step 31, the

aggregate stream is written across the target locations in the order chosen in step 28. An optional

map of these target locations may be produced for later access by other aspects of the present invention that may not contain the algorithmic knowledge to determine those locations without such a map.

5   FIG. 7 is a flow diagram illustrating a method by which the hidden, stored content is retrieved, for example information previously hidden in secure or non-obvious locations as shown in FIG. 6. In this process, the information is retrieved and reassembled into its original form and provided as needed to other system components. In determining the possible locations where such information could be hidden, there are, for example, two possible initial sets of
10   actions 33; either obtain the map information previously hidden according to step 28 of FIG. 6, or generate a valid retrieval map as an equivalent of the storage map by incorporating the same algorithmic storage logic as retrieval logic, for example the process employed in **FIG. 6**: determine all possible locations 27, select the chosen locations and ordering 28, and create the retrieval map equivalent of a storage map.

15   Low-level operating system interfaces are accessed, and device level access is initialized 34 at a level far below the normal file system interfaces, such that the device may be addressed in any and all valid raw physical locations, whether inside or outside the standard file system. The map or map information obtained above at step 33 is used to determine the ordering or
20   reading and the read locations, and these locations are read in order 35. The items read are concatenated in the order read to re-create the original multiplexed interleaved stream. If decrypted previously, the decryption key is read, either from the map 33 or from a predetermined location which may be at the beginning of the encrypted stream 43 (see FIG. 8), at the end of the encrypted stream 42, at a predetermined offset within the stream 44, or subdivided and hidden at
25   predetermined offsets 47,48,49,50 within the encrypted stream 45, and is itself decrypted at step 36 of FIG. 7. The stream itself is decrypted 37 as desired. The stream is de-multiplexed into its component original streams 38. Each component stream is subdivided into a number of segments of a predetermined number of bits in length and each segment is then de-interleaved 39 into its original component input stream. Each such stream is then written to the file system 40 or
30   otherwise provided to the system.

Returning to **FIG. 4** the Intrafile space 20, or space within the bounds of a file, is space that is usually specified as "unused" or "reserved for future use" in the specifications for the file

5   or stream types. The following list of published specifications represent a sampling of those researched to determine space utilization within various types of files:

- ❑ "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format", Matt Pietrek, March 1994
- ❑ "BMP Format: Windows Bitmap File Format Specifications", Wim Wouters, May 2000
10 ❑ Appnote.txt from the PKZip Website
- ❑ The ISO-ITU JPEG standard in a file called itu-1150.ps
- ❑ CRYX's note about the JPEG decoding algorithm. Copyright 1999 Cristi Cuturicu.
- ❑ Inside Windows Cabinet Files by Sven B. Schreiber

15   Using this research data, and proprietary data collected manually by examining many available file types, the present invention embodies a set of programmatic rules that represent techniques for placing data within all the known safe locations (see FIG. 6, step 27) to store protected (interleaved and/or multiplexed and/or encrypted) data in all tested file types, and once hidden, the present invention provides a similar inverse set of capabilities (see FIG. 7) that

20   provide mechanisms to find the hidden information (see steps **33 34 35**), extract it (see steps **36 37 38 39**) and provide the decrypted, de-interleaved data to the requestor at step 40 of FIG. 7.

The following code example illustrates an embodiment of the invention described above and the programmatic rules illustrated above and with reference to **FIG. 6**. Each type of file (for

25   instance text files, jpeg photographs, GIF web images, executable "exe" or PE files, any and all types of files known to the operating system), have specific rules within this invention associated with them. The code example below shows the logic used to determine the available free space within a given file. One of the parameters is a call-back process (writeMapLocation) which creates a list of available locations in the form of a map structure (sometimes called a "raw"

30   map). The second parameter is the current MapRawList to which the informative list is to be written. The method used to determine the byte locations to pass to writeMapLocation varies for each file type (BMP, EXE, etc).

34

```
     CBMPFile::GetMapLocations(
         void (*writeMapLocation) (unsigned long,unsigned long, bool, bool,
5    bool, MapRawList_t **),
         MapRawList_t **rawMapTail
     )
     {
         unsigned long i;
10       unsigned long pos = startLocation + STARTOFPALETTE +
     (PALETTE_ENTRY_SIZE - 1);

         for (i=0;i<paletteEntries;i++)
         {
15           (*writeMapLocation) (pos, 1, false, true, true, rawMapTail);
             pos += PALETTE_ENTRY_SIZE;
         }


     }
20

     //
     //  FUNCTION: WriteMapLocations(unsigned long offset, unsigned long length)
     //
25   //  PURPOSE: Added the given locations to the RawMapList
     //
     //  COMMENTS:
     //
     //
30   void WriteMapLocations(
         unsigned long   offset,
         unsigned long   length,
         bool            isNonZero,
         bool            isAlwaysFindable,
35       bool            isInsideFile,
         MapRawList_t ** rawMapTail
     )
     {
         BYTE flags = 0;
40
         if (length == 0)
             return;

         if (isNonZero)
45           flags |= ISNONZEROFLAG;
         if (isAlwaysFindable)
             flags |= ISALWAYSFINDABLEFLAG;
         if (isInsideFile)
             flags |= ISINSIDEFILEFLAG;
50
         (*rawMapTail)->offset = offset;
         (*rawMapTail)->length = length;
```

35

```
(*rawMapTail)->flags = flags;
(*rawMapTail)->link = (MapRawList_t *) malloc (sizeof(MapRawList_t));

*rawMapTail = (*rawMapTail)->link;
InitMapRawEntry(*rawMapTail);
}
```

5

10

In another embodiment of this invention illustrated in FIG. 9, content is placed in various locations and then protected using a technique referred to as translocation, a process that is described in further detail below. Prior to discussing the concept of translocation, it is necessary to first describe the nature of such locations for the placement of such information. Such information may be executable content such as a Windows program, for example notepad.exe, or may take the form of other content, for example, a text file, a movie, or an audio file or music. The file system consists of storage space on one or more devices and a table of contents or directory that provides locations and offsets. There are multiple embodiments of this invention with alternate strategies for placement which may be used individually or in combination. Note that content may be placed as follows in whole or in part, since hiding even part of complex content may render the remainder useless, such that the first 25% of a given content type can be hidden and the remainder is made secure by the lack of the hidden part, even though the remainder is accessible.

In one such implementation, content may be placed within the file system **65** but hidden between the files **56** in space, for example, that is created by the fragmentation of predetermined storage blocks on the storage media such that the files visible in the file system do not entirely occupy the space allocated for them. Such content is placed in unused between-file fragmentation space within the bounds of the file system **56** such that its location is unknown to the table of contents **54** so that no file system access at the file level will be able to locate or access the files. This type of information hiding may require the information be subdivided into small parts and hidden in multiple smaller locations, since the available space between files may be fragmented.

In another embodiment 66 such content may be placed outside the file system entirely 59. In this implementation, the amount of contiguous available space is larger and thus such a file may be placed in contiguous locations, however note that such a file may in fact still be

5   subdivided and placed into multiple disordered discontiguous locations for added security even in the abundant contiguous space in such extra-file system 59 locations.

In an alternative embodiment 67, the content is placed partly between the files within the file system 62, and partly in space outside the file system, namely the extra-file system 63.

10

The concept of translocation as implemented in this invention and as illustrated in **FIG. 9** is described with reference to examples **65, 66 and 67**. Assuming that the apparent target is a hacker's tool such as "ProcDump.exe" and the translocation replacement is a stub executable whose sole instruction is to exit, any attempts to execute this hacker's tool, such as by double-

15   clicking on it with a mouse, would result in the execution instead of the stub, which would immediately exit, such that the execution of ProcDump would appear to have failed to an outside observer with no apparent reason why. The actual mechanisms by which this process operates are as follows. The protected content is copied from its former location 55 to a new location 56; it may be optionally encrypted during the copy process if desired. In the present example this

20   location is actually a series of noncontiguous smaller locations that the content is subdivided into, between files of the file system in the space created when file system blocks are fragmented due to partial usage. These blocks, when used, are marked in the file system's records so they will not be inadvertently overwritten or re-used, but they do not have a corresponding entry in the directory system so they are not accessible from the standard file system interfaces. The

25   former location 55 is populated with a file whose attributes are identical with the protected content in terms of name, size, external appearance, but whose behavior or contents differ as desired (in the above example, ProcDump is replaced with a stub that exits). Attempts to execute "ProcDump" are made but they access the former known location 55. The translocation system can at any time retrieve the actual contents from the new location 56 and either repopulate them

into the former location **55** or provide them as needed to the other components of the present invention.

Similarly in examples **66** and **67**, the locations that are populated with the translocated

5   content (in this case the real "ProcDump.exe" we're hiding) are either outside the file system entirely **66**, or, in the case of example 67, partly within the fragmented between-file space and partly outside the file system.

Note that in an alternate inverse embodiment of this invention, the original file is not

10   moved at all **55** but rather the translocation replacement file is placed into the new location **56**, and the file system's pointers 57 are temporarily updated to point to the translocated replacement file. Note that locations outside the bounds of the file system, for example location **59,** may be on the same media as the file system or on entirely different media , for example, random access memory, rewriteable storage, network storage, or any other viable storage medium accessible to

15   the system.

An example process used to create a translocation replacement file is now detailed with reference to FIG. 10. For continuity the example above is referred to, where the original file is "ProcDump.exe" and the translocation replacement is "stub.exe" which does nothing other than

20   exit (of course any file of any type may be replaced by any other file of the same or different type, as desired) 75. The ProcDump file is first scanned and its attributes recorded; any icons or other resources are copied and duplicated 68. The ProcDump file is copied at step 69 to various predetermined storage locations, for example locations 56, 69, 62, and 63 of FIG. 9. Optionally to ensure added security, the original contents of ProcDump are zero-filled 70 and deleted in

25   entirety 71 from the media, while bypassing the file system so that the directory entry and pointers remain intact. The original location is used as the location and bounds for the translocation container 72, and this container is then populated with the icons 73 and other attributes 74 of the original "ProcDump.exe", and the container is then populated with the logic and contents of "stub.exe". Thus any attempt by an unauthorized individual to execute

30   "ProcDump.exe" results instead in the execution of "stub.exe", and this persists even if the file

known as "ProcDump.exe" is copied elsewhere, since the content has been replaced at a physical level.

With reference to FIG. 11, in certain embodiments, there may arise circumstances where an authorized entity has a valid need to access content which had previously been translocated as above. Operating system interfaces for file access can in this case be monitored, and attempts by an authorized entity to access the translocation container 76 result in retrieval of the original target 77 from storage locations. If encrypted as part of the storage process, decryption is performed on the content 78. An execution environment appropriate to the content type 79 is invoked on behalf of the requesting entity (for example, if the protected content were "readme.txt", a text file, the application "notepad.exe" might be launched). The retrieved content "readme.txt" is then provided to the execution environment 80, and the requesting entity's needs are met ubiquitously.

As explained above, translocation is defined as the ability to provide ubiquitous redirection, which may be used for both the hiding of information, and for the purpose of defending against attacks by disabling the opponent's access to the necessary reverse engineering tools. Translocation may be embodied in a system that actually moves content, or in a system that redirects access to content without moving it. For example, in the case of moving content, an individual's intent on reverse engineering a protected system may wish to run the Visual C++ development tools to attempt to debug the running system. When the protective system is invoked, among the first things it does is translocate all threatening tools it finds, such that Visual C++ is moved from its old location 55 to a new location 56 (see FIG. 9), and the contents of location 55 are replaced with an executable that does nothing but exit when run. Thus when an attempt is made to run the executable file for Visual C++, the file that is actually run is this stub executable that does nothing useful.

An example of translocation that redirects without moving content is similar. With reference to FIG. 23, such a mechanism employs a connection to the operating system interfaces 137 for, in this case, file access, and when an attempt is made to run Visual C++ at location 55

39

(see FIG. 9), the call is monitored and intercepted at steps 138, 139, and the executable file that is actually run 140 is the replacement stub file 56. This replacement stub file can do far more than just exit; an example is an embodiment of this invention in which the replacement file is a crippled version of the desired target file 55. In order to further obscure what is happening, care

5   is taken in this example that when the replacement or redirected file is invoked (for example FIG. 11) to touch 141 the desired file 55 so that any file system monitoring tools that may be running will see the expected access 55. Note that as in examples 66 and 67 of FIG. 9 there are embodiments of this invention in which the redirected or moved content resides wholly or partly outside the file system 59, 62, 63, and embodiments in which the redirected or moved file does

10   not reside in contiguous locations but rather in two or more subdivided locations 62, 63. In one such embodiment, the translocated content is stored in the fashion that an M-bit watermark 12 is stored 31, across multiple M-bit locations with no regard for contiguity, and later accessed by means of the methods described above in association with FIG. 7.

Note that translocated content leaves no obvious clues; the process used to create 73 these

15   substitute or redirected files as in the example FIG. 10 insure that the replacements have all the proper attributes, through steps 68 and 74, including all icons, size and date attributes, and all other properties of the original. Also note that the above example was related to an executable program file, but there are other embodiments of this invention. In one such embodiment, the

20   content is audio, and when invoked in the process of FIG. 11, the act of execution causes the concurrent invocation 76 of an appropriate audio player/helper application 79. In another embodiment of this invention, the content type is a digital video stream, a popular movie title. In this case, the execution environment 79, when invoked 76, is a digital video player helper application. All digital content types are therefore supported by this aspect of the invention.

25

Another embodiment of this invention as exemplified in FIGs. 12, 13, 14, 15, and 16. This embodiment relates to a set of mechanisms that operate to tokenize and obfuscate (see step 83 of FIG. 12, reference 88 of FIG. 13 and step 92 of FIG. 14) content of all types (see step 98 of FIG. 16, below) in order to eliminate trivial observational analysis, and in the case of executable

30   content, to greatly increase the difficulty of unauthorized debugging. This embodiment also

serves to prohibit the modification of all types of content, since the tokenized obfuscated content **89** cannot be modified using standard editing/modification methods due to its proprietary tokenized formatting. In the case of executable content, disassembly is also prohibited by this process since the resultant output **84, 89** is no longer standard assembly language.

5

For example, with reference to FIG. 12, digital content 82 may be tokenized according to any of a number of standard tokenization mechanisms 83, and the resulting tokenized content 84 is stored (see FIG 13, step 89). With reference to FIG. 15, the stored tokenized content 93 can be later be retrieved and subsequently reconstituted and executed **94**, provided an execution

10   output 95 that is the same as that which is originally intended.

With reference to FIG. 13, the stream of digital content to be tokenized and obfuscated **82** (see **FIG. 12**) is presented. The digital content is read and its type is determined **86**. The system and method of the present invention preferably recognizes all existent digital content/file/stream

15   types; in the case of this example the file type is determined to be an executable or Windows "PE" file conformant with the specifications found in "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format", Matt Pietrek, March 1994. The content is parsed **87**, with a lexical parser similar to those found in many compiler front-end mechanisms. Portions of the content are replaced with tokens **88** that bear an appropriate lexical relationship **91**,

20   understood to the mechanisms of this invention, to the content and the context. In one example the token replacement may be fixed; for example the assembly language MUL or multiply operator is replaced with the token ^. To further complicate this example, the token replacement may be variable, for example based on location, such that the MUL operator's token is ^ if it occurs in the first 50 lines of assembly code, otherwise it is #.

25

Details related to the substitution of tokens are provided at FIG. 14. The content is parsed at step 90, as described above in FIG. 13, step 87. Lexical boundaries of the parsed content are identified 91, and the replacement is performed. In other words, using the English language as an example, if one were tokenizing the sentence "My dog does not understand my

30   dogma." it might be appropriate to replace the term "dog" with the token "*", but it would be

wrong if we also made the same replacement within the word "dogma" and turned it into "*ma" because the context and lexical meaning of "dog" and "dogma" are different despite the fact that the first three characters are identical. A context free search would find them to be the same; "dog" matches "dog" and matches the first three characters of "dogma" but since the meaning is

5   different, the system must be intelligent enough to do more than match the appearance of an item; the item's meaning and contextual relationship must be understood. Thus it is not a simple context free blind replacement such as doing a global replace edit using Microsoft Word; the location and meaning of each item, and its relationship to items before and after it are all relevant to the substitution logic used to tokenize it.

10

Returning to FIG. 13, the tokenized content is written out **89**, and may then be interleaved, multiplexed, encrypted, and/or hidden as illustrated in the previous examples described above.

15   With reference to FIGs. 15 and 16, at a later time, as needed, when it is time to execute this content, the tokenized content 93 is located and extracted at step 97 (if it was indeed interleaved, multiplexed, encrypted, and/or hidden as described above). The content type is determined at step **98**, and the tokens are parsed and converted back into standard executable code **99**. The content may then be re-obfuscated **100** by applying known variations on standard

20   assembly language which serve to confuse debugging and disassembly tools. It may then be executed in an appropriate execution context **101**; in the case of executable "PE" program code, that context is the operating system itself to be executed **102** upon the processor **5** (see FIG. 1).

In the example below, this invention replaces standard assembly language elements with

25   permuted assembly language which has attributes that cause disassembly utilities such as, for example, the popular disassembly tool IDA Pro, sold and distributed by the Belgian firm *DataRescue*. Such tools depend on assembly language being formed and structured in specific standard ways; the enhanced assembly language generated by this invention offers the same logical function as the code it replaces but is resistant to disassembly as shown in the example

30   code illustrations below.

The first such code example below illustrates this invention's insertion of jmp statements to instances of the following assembly language instructions: inc, dec, call, jmp, and push

5

```
Convert this:  0000: 90 nop 0001: FF inc
To this:        0000: EB FF jmp 0001 0002: inc
```

10   For example, this embodiment changes instances of "jumps" to (push and return) calls:

```
Convert this:  stmt: JUMP2V(addrjmp)   "\tjmp\t%0\n"  3
To this:        stmt: JUMPV(addrjmp)    "\tpush\t$%0\n\tret\n"  3
```

15

For example, jumping into the middle of an instruction to confuse all disassemblers:

```
erp:   mov   ax,0FE05h
       jmp   $-2h
20     add   ah,03Bh
```

Another code example of the same class of techniques used by this invention :

25
```
B8 05 FE EB FC 80 C4 3B   mov   ax,0FE05h   ; ax=FE05h
B8 05 FE EB FC 80 C4 3B   jmp   $-2         ; jmp into '05 FE'
B8 05 FE EB FC 80 C4 3B   add   ax,0EBFEh   ; 05 is 'add ax'
B8 05 FE EB FC 80 C4 3B   cld              ; a dummy instruction
30   B8 05 FE EB FC 80 C4 3B   add   ah,3Bh    ; ax=2503h
```

Note that the "add ah,03Bh" command is instantiated to insert the value 2503h into location ax. By adding five bytes (as opposed to simply using 'mov ax,2503h') this code will defeat all known disassemblers. Even if the instructions are disassembled properly, the value of

35   ax will not be known, so every int call after this point will not be commented properly, as long as the system never moves a value into ax. This embodiment of the invention can conceal the value from the disassembler by using 'add ax' or 'sub ax' whenever possible. Thus any value can be put into ax.

40   This invention, of course, must make such substitutions in an automated fashion; the code example below illustrates such programmatic assembly language substitution:

43

```
/* Output the anti-disassembly code */
/* Based on the following code
print("mov ax,0FF05h\n");
print("jmp short $-2h\n");
print("mov ax,0FFFFh\n");
print("jmp short $-07eh\n");
*/
{
        unsigned char randomBytes[10];
        int i;
        char buf[100];

        for (i=0;i<4;i++) {
                randomBytes[i] = rand() % 256;
        }

        sprintf(buf, "\t.byte 0x66, 0xb8, 0x05, 0x%.2x\n",
randomBytes[0]);  /* mov */
        print(buf);
        sprintf(buf, "\t.byte 0xeb, 0xfc\n");          /* jmp */
        print(buf);
        sprintf(buf, "\t.byte 0x66, 0xb8, 0x%.2x, 0x%.2x\n",
randomBytes[1], randomBytes[2]);  /* mov */
        print(buf);
        sprintf(buf, "\t.byte 0xeb, 0x%.2x\n",
randomBytes[3]);        /* jmp */
        print(buf);
}

emitcode();
```

In an alternative embodiment of the above aspect of the invention, and a variant example, the inventive system and method, after having tokenized and obfuscated the content and optionally interleaved, multiplexed, encrypted, and/or hidden it, later, as needed, when it is time to execute this content, the content is located and extracted (if it was indeed interleaved, multiplexed, encrypted, and/or hidden), parsed, content type determined, the tokens are parsed and execution occurs in lockstep with the conversion to executable content so the reconstituted content is never written to a file or provided to any entity in the system, but is rather executed on the fly within a custom execution context 101 (see FIG. 16) or custom interpreter **101**. Note that "content" may be any digital content; executable program code, audio, video, digital documents, and the "execution content" is constructed to execute the content. The meaning of "execute"

varies depending on the content; for example audio or video would be executed on an appropriate audio or video player, documents presented in an appropriate viewer, application programs and games run.

5   An embodiment of this invention may generate for example instances of the variant assembly language as illustrated in the example above, and thereby be resistant to disassembly, and may also be made more difficult to debug by defeating automatic disassembly tools using obfuscated assembly language programming techniques, for example inappropriate not-used jumps into the middle of instructions. Such obfuscation, or similarly effective methods

10   accomplished by other means, enhance the security of the invention. Note that this is in addition to the inherent security of running within an interpretive environment. The interpreter operates as a shield from debugging and reverse-engineering tools. The interpreter serves as a layer of abstraction between the protective invention and the real operating system. The values found in system memory and registers will not be directly related to the logical flow of the interpreted

15   program; they will show the debug state of the interpreter itself instead, and that will make assembly language debugging very difficult.

       In another embodiment of this invention described with reference to **FIG. 17** and **FIG 18**, a protective system for digital content, or any running software application or system of any

20   kind on any platform, is itself protected from being debugged, monitored, logged and understood by an invention mechanism which creates carefully targeted and tuned system activity, or "saturation" activity. This activity causes an instrumented or debug-enabled computer system to generate large volumes of debug, log, and/or monitor-tool traffic unrelated to the protective logic. For example such traffic can make a log that would have been 15 kilobytes grow to be 150

25   megabytes. Monitoring/logging/data watching debug techniques are easily overwhelmed by this approach. One example of such a logging monitoring tool and it's usage is Filemon, an excellent freeware tool which logs system file activity. When exposed to the saturation traffic **110**, the Filemon event log can grow to be orders of magnitude larger than it would otherwise be. Events of interest to one debugging or reverse engineering the system are therefore lost in the process.

30

This targeted saturation embodiment of the present invention operates as follows. The protection by saturation of a system or application first depends on understanding the nature of the normal system traffic generated by that application. Therefore, with reference to **FIG. 17,** the protected entity must first be analyzed as in step **107**. The protected entity is executed on a

5   system that is running the saturation profiler tool **104**. This tool profiles activity **104** in such ways that classes of activity are monitored (for example SCSI calls or registry calls or file opening) and statistics are gathered (for example, scsi calls logged during the execution of program material to be protected). For example, *400 file opens, 3500 reads of 2048 bytes each, 120 query commands*. All aspects of system utilization are monitored and logged and categorized

10   by type and frequency. This forms a profile of activity for the program material. This profile is encoded in a fashion readable by a later process of this invention (**FIG. 18,** described later in this document), and written to a "saturation list", along with a tuning profile **105** with detailed encoded instructions **106**. These instructions specify the desired traffic types and volumes, for example to mask the SCSI traffic, in one embodiment, the present invention is directed to

15   generate 4000 file opens in similar drive locations and sizes, 30,000 reads, 500 query commands.

As described in **FIG. 18,** the invention which actually generates the directed saturation traffic may first open the saturation profile **108**, decode the instructions as required, determine which types of traffic are desired (for example network traffic, or as in the example above SCSI traffic), communicate with the appropriate saturation engine (as above, the scsi saturation engine

20   would be used in this example; each such entity may be used individually or in combination, such as for example doing both SCSI and network saturation) **109**. The saturation engine then executes the required commands **110** and **FIG. 19,** (see below for details) and generates the appropriate levels of traffic.

25

The functioning of an individual instance of a saturation engine **116** is shown in **FIG. 19**. The SCSI example from above provides an illustration to one skilled in the art; the SCSI interfaces are utilized and an event driven mechanism is created, where the first logical step is to wait on the event of either a command completion or a new external request to issue a command

30   **112**. Upon awakening, if a command is pending (a SCSI file open, for example, as the next

saturation command in the desired saturation list), it is executed **113**, and synchronously waited upon if desired **114** with varying next-step results optionally depending on completion status. If normal completion, the process executes a hard sleep for a predefined interval if desired (to throttle back activity) **115**, and then sleeps again waiting on the events as in **112**. This is indeed a loop and would be infinite if the queue of commands were infinite, however being event driven, the loop suspends execution after the last command is consumed and is optionally swapped out, eliminating system resource utilization until again needed. The throttle-back sleep allows the saturation system to selectively control its utilization of system resources dynamically, for example to avoid monopolizing system resources when they're needed for more important activities. The ability to be throttled back is controlled by the process of the invention as needed to reduce saturation traffic in specific ways at specific times, and may be overridden programmatically by other invention embodiments within the protective system if they determine they need more resources for any reason.

All individual saturation engines are controlled by a saturation scheduler as shown in **FIG. 20**. The scheduler opens, decodes, and reads (parses) **117** the saturation profile and system settings directions from the saturation list previously described. The necessary saturation engines are polled, **118** launched if not already present, and the engine specific commands (for example SCSI commands as above) are queued to the saturation engine's **123** main scheduling loop. The underlying process driving the command queue mechanism is event driven and clock driven, with saturation engine tasks being fed commands at predetermined rates. The command feeder process is itself event driven, sleeping and waiting **119** upon the event of commands entering the queue, issuing the command **120** with dynamically controllable command frequency and adding additional sleep time commands to the payload so the saturation engine knows how much additional sleep over and above the event queue events is required (this is the throttling mechanism as described in the paragraphs above), and monitoring the effect on the system to determine if the throttling amount and the command queue depth and speed are appropriate to the task. This main scheduling loop **123** would be infinite if not event driven, however since it is event driven (as the individual saturation engine loops are) when the queue of commands is empty, the system is quiescent, suspended, and optionally swapped out. Upon overall

47

completion, the scheduler exits **123** and may optionally kill all the individual saturation engines previously spawned.

In another embodiment of this invention as shown in **FIG. 21**, a filter, shim, device driver

5   extension, or substitute device driver is inserted into system interfaces, interposing itself **125** between the original driver or interface and all other entities by stealing inputs directed towards those interfaces, reattaching any previously attached entities to the public "subsumed interfaces", optionally passing through or modifying the traffic to those interfaces, optionally logging traffic, thus subsuming the "public face" of such interfaces. An example would be to take over the

10   interface to the system "beep" function. Every time a system "beep" (the annoying noise the PC speaker can make at power up on many Personal Computer systems) is requested, the shim steals the command. In this example, if the requesting process is your email program, the beep is passed through, and the system beeps. If the requesting entity is a disallowed entity, like an equally annoying pop-up browser window, the beep may be thrown away and thereby

15   suppressed. Note the vulnerability of such an interface shimming techniques in its simplest form is that another such "imposter" shim intended to compromise such a "protection" shim could be inserted after (or before, or both before AND after it, to allow it to be bypassed entirely at will, depending on the intent) the protection shim, thus obviating the utility of such a mechanism. In other words, the shim itself can be monitored or subverted if it in turn is shimmed. Therefore this

20   invention compensates for that vulnerability by continually reconnecting. The process as shown in **FIG. 21** initiates by first finding the system interfaces it intends to subsume and uses the lowest possible level of interface; interface use is performed based on that low level information rather than using higher level abstractions made available by the operating system. The interface's external interface functions are subsumed by the shim **125**, any commands received

25   while impersonating the interface are optionally either passed through, modified or discarded (the system may desire to do any of those things, for example if authorizing by PID, a read access might be thrown away of the requesting PID were believed to be a security threat like a debugger) **126**. Alternatively, the system could transparently pass all requests through 126 and optionally offer an undocumented other interface so a knowing programmer could access system

30   functions through the shim directly **126**, bypassing system interfaces and associated interface

monitoring tools. For example as part of a broad throttling process, the process may optionally

sleep between subsumed-interface-commands **127** thereby retarding public interface access, thus

providing reduced system resource usage as desired to specific entities on the system as needed

(for example to starve a reverse engineering tool and reduce its utility). Once a number of such

5   commands have been processed and time intervals optionally slept by the process, it detaches

from the operating system interfaces and immediately reattaches **128** again at the lowest level;

this to ensure that it has not been compromised by another shim inserting itself before or after it.

This reattachment loop **129** may be infinite, the shim may be left in place indefinitely to exit

upon system shutdown, and optionally not reconnect at next reboot, effectively thereafter

10   disappearing from the system.

       In the code example below, this dynamic-reconnection mechanism of the present

invention manifests itself as a process that attaches to the first location directly at the interface

level, and forces all subsequent shims of any other kind to attach themselves after the invention

15   by continually reattaching in the first position:


```
// find the bottom of the bottom of the OS-Interface ShimList; AutoReAttach is placed
//at the top of the ShimList. If an authorized request is received, we use the saved location of the //bottom
of the OS-Interface ShimList to bypass anyone who might be Attached in between
//If an unauthorized request is received it is passed down the ShimList normally.
//The Attach and reAttach logic keeps the _ Attach at the top of the ShimList.


// Install and remove a dummy SystemInterface Attach in order to get
//    the address of the last Attach in the OS-Interface ShimList
s_pPrevAttachDummy = ANYINTERFACEMgr_InstallSystemInterfaceApiAttach(FnAttachDummy);
ANYINTERFACEMgr_RemoveSystemInterfaceApiAttach(FnAttachDummy);


// Keep going until we get to the OS-Interface itself
apAttachs[0] = s_pPrevAttachDummy;
wldAttach = GetAttachId((BYTE *)*(apAttachs[0]), NULL);
idxShimListDepth = 1;
while (wldAttach != ANYINTERFACEMGR_VXD_ID)
{
    // Remove all of the Attachs we have found so far
    for (ii = 0; ii < idxShimListDepth; ii++)
    {
        ANYINTERFACEMgr_RemoveSystemInterfaceApiAttach(*(apAttachs[ii]));
    }
```

```
      // Add and remove a dummy Attach to get the pointer to
      //   the next Attach in the ShimList
      s_pPrevAttachDummy =
ANYINTERFACEMgr_InstallSystemInterfaceApiAttach(FnAttachDummy);
      ANYINTERFACEMgr_RemoveSystemInterfaceApiAttach(FnAttachDummy);
      apAttachs[idxShimListDepth] = s_pPrevAttachDummy;

      // Now replace all the Attachs we removed above
      for (ii = idxShimListDepth - 1; ii >= 0; ii--)
      {
          ANYINTERFACEMgr_InstallSystemInterfaceApiAttach(*(apAttachs[ii]));
      }

      // Get the ID of the most recently found Attach
      wIdAttach = GetAttachId((BYTE *)*(apAttachs[idxShimListDepth]), NULL);

      // Increase the depth by one for the next pass
      idxShimListDepth++;
    }

    // Remember the address of the final OS-Interface  "Attach"
    s_pAnyInterfaceAttach = s_pPrevAttachDummy;


    // Install our Attach at the end of the ShimList
    if (s_dwSiDct == 0)
    {
        s_pPrevAttach = ANYINTERFACEMgr_InstallSystemInterfaceApiAttach(RchwyAttach);
    }

static void FixAnyInterfaceShimList(
    //
    //
    //
    )
    {
      // Install and remove a dummy SystemInterface Attach in order to get
      //   the address of the last Attach in the OS-Interface  ShimList
      s_pPrevAttachDummy = ANYINTERFACEMgr_InstallSystemInterfaceApiAttach(FnAttachDummy);
      ANYINTERFACEMgr_RemoveSystemInterfaceApiAttach(FnAttachDummy);

      // If we aren't the last Attach in the ShimList, remove our Attach and
      //   then reinstall us to get us back at the end of the ShimList
      if (RchwyAttach != *s_pPrevAttachDummy)
      {
          ANYINTERFACEMgr_RemoveSystemInterfaceApiAttach(RchwyAttach);

          s_pPrevAttach = ANYINTERFACEMgr_InstallSystemInterfaceApiAttach(RchwyAttach);
      }

      return;

    } // End FixAnyInterfaceShimList
```

In another embodiment of this invention, described with reference to FIG. 22, such an attach and re-attach strategy is implemented for the purposes of feeding spurious or saturation traffic into an opponent reverse-engineering tool. In other words, this invention may be used to isolate and defeat certain reverse engineering tools. For example, if the tool FileMon (an excellent reverse engineering tool distributed by SysInternals.com) were in use, it would effectively monitor all usage of the filesystem and record all access in detail. If it were desirable to hide access from such monitoring tools, one such invention use for example would be to isolate FileMon by attaching one shim before it, and one after it, and having each shim continually reattach itself. If each such shim had a data connection to each other bypassing FileMon it would be trivial to shunt all traffic around FileMon, effectively causing it to record nothing. In more subtle usage examples, selected traffic could be hidden from FileMon in this fashion, while spurious saturation traffic was directed through it.

In this embodiment, as above, a filter, shim, device driver extension, or substitute device driver is inserted into system interfaces in this case, interposing itself at step **131** between the reverse engineering monitoring shim and the rest of the system, thus apparently subsuming the role of the operating system interface and providing false and misleading data **132** to the monitoring/reverse-engineering shim/tool. The vulnerability of all such interface shimming techniques in their simplest form is that another such shim intended to compromise such a shim could be inserted after (or before, or both, depending on the intent) this process at any time, thus obviating the utility of such a mechanism. Thus, this embodiment of the invention includes a re-attachment mechanism **134** which guarantees a specific attachment location, in this case directly before the opponent reverse-engineering/monitoring shim, as specified by the invention's user. This is accomplished by repeated automated re-insertions **135** into the interface chain. Such reinsertions are done in a fashion that does not impede function by waiting a number of time units **133** between issued instructions. Thus this embodiment of continual-interface-reattachment can eliminate the threat of device redirection and monitoring tools being used to subvert the system.

In another embodiment of the present invention, as illustrated in **FIG. 23**, ubiquitous redirection of operating system interface access is employed to prevent the execution of, or access to, content that is disallowed, or to redirect access to other content in a manner that is transparent to the accessing party or process. As above, this embodiment of the invention

5   connects to the appropriate operating system interfaces at step **137**, executing the reconnection logic as needed as in **FIG. 21** and the description above. Calls to the interface are monitored **138**, and when appropriate, intercepted **139**. For example, if a tool such as FileMon were discovered on the system at the time of the invocation of this embodiment, it would be logged as an "access to monitor" and when it was accessed **138**, it would be noted, and access would be

10   redirected from the FileMon operation to a different executable **140**, in this example an executable that does nothing but exit. At the same time this redirected executable was launched **140**, the originally intended executable is touched **141**, such that any other monitoring tools would show the access. Thus the individual intent on reverse engineering would launch FileMon and it would exit immediately **142**. The individual might use other tools and discover that

15   FileMon did indeed launch (file system access to the original file will be logged as though it was launched).

The code example below illustrates the invention discussed above in conjunction with **FIG. 23**; a means of redirecting access **140**, for example, from one executable **138** to another

20   **139** ubiquitously:

```
// If the access is one that the system wishes to disallow
//    and redirect, and a stub exe has been loaded,
//    point it at the stub file instead
25      if ( ((DWORD)(-1) != s_idxStub)        && // stub loaded
        (!fPidMatch)              &&          // choose to disallow this one
        (fIsExec))                            // and it is a .exe
                {
        ii = s_idxStub;
30      }
```

The code example below illustrates the invention discussed above in conjunction with **FIG. 23**; in this case the code example is the do-nothing stub executable that replaces access to

35   the disallowed executable(s).

```
     int APIENTRY Main(
     //
     //
 5   //
         HINSTANCE   /* hInstance       (unused) */,
         HINSTANCE   /* hPrevInstance   (unused) */,
         LPSTR       /* lpCmdLine       (unused) */,
         int         /* nCmdShow        (unused) */
10   )
     {
         // Do nothing
         return 0;

15   }  // End Main()
```

In another embodiment of the present invention, a protective entity is created; such entity

20 operates as an independent protective agent and secures all protected content from unauthorized

access. As depicted in **FIG. 24**, this entity, referred to as an "assassin", may be programmed to

have multiple functions. For example, the assassin upon initialization 144 first determines how

many other assassins and other protected entities are present 145. System authorization

functions are utilized **146** as depicted in **FIG. 25, FIG. 26** to establish the correct identity of all

25 processes on the system at all times. The assassin scans the system for the presence and

execution of threat-entity-instances, such as debug tools like ProcDump and FileMon and even

developer tools like Microsoft's Visial C++ **147**. It also uses the functions detailed below to

track the process or thread exit of any other entity including other assassins **148**. Upon

determining intrusion has occurred (debugger running, unauthorized exit of any other assassin

30 protective entity, any changes or modifications **149** made to code or system components in any

way within the system by any unauthorized entity, presence of ICE or other debugger) an exit

condition is set up in which this assassin, and other assassins, and other system components will

exit **150** based on either noticing that another has indeed exited or by passing a signal event

between components of the system. In some cases an exiting assassin will kill **150** other system

35 entities as a means of accelerating overall system component exit.

In the code example below, a first embodiment of the assassin process determines the

identity of another assassin process (this is a two-assassin example) and instances **146**, and

monitors them for exit conditions **148**. Upon an exit condition, this embodiment attempts to kill

other assassin processes and then kills itself **150**.

```
        //  Wait for a target entity to exit
 5      static bool WaitAndDeleteInstance(
        //
        //
            DWORD   in_dwIdentWaitProc1, //  1st proc to wait for
            DWORD   in_dwIdentWaitProc2, //  2nd proc to wait for
10          DWORD   in_dwIdentKillProc,  //  proc to kill if proc 1 exits
            char*   inp_szFn,            //  instances to delete
            char*   inp_szFnFk,          //  more instances to delete
            char*   inp_szFnDel          //  add'l instance to wait for  (NULL for assassins)
        )
15      {
            HANDLE    ahProc[2] = {NULL, NULL}; //  handles to wait on
            DWORD     dwRes;                    //  result from wait
            int       ii;
            char      szFnWait[MAX_PATH];       //  instance to wait for
20          char      szFnDel[MAX_PATH];        //  instance to delete
            bool      fTargetInsOpenFailed = false;
            HANDLE    hTargetIns;
            char      szIsDel[MAX_PATH];
            char      szTargetIns[MAX_PATH];
25          strcpy(szTargetIns, inp_szFn);
            strcat(szTargetIns, "target.inf");
            strcpy(szIsDel, inp_szFn);
            strcat(szIsDel, "targetEntity");
        */

30
            //  Open handle to the 1st proc.  This will be the 2nd assassin entity
            ahProc[0] = OpenEntity(ENTITY_ALL_ACCESS,
                        FALSE,
                        in_dwIdentWaitProc1);
35          if (NULL == ahProc[0])
            {
                //  If we can't open this entity handle, then something is definitely
                //    wrong, so kill the redirected (target) entity if there is one
                if (0 != in_dwIdentKillProc)
40              {
                    KILL_ENTITY_FROM_IDENT(in_dwIdentKillProc);
                }

                //  Delete the instances and return
45              DelTree(inp_szFn);
                DelTree(inp_szFnFk);
                return false;
            }

50
            //  If no other entity was specified, then the current entity must be one
```

```
//   of the assassin entities
if (0 == in_dwIdentWaitProc2)
{
    // Wait for the original entity
    WaitForSingleObject(ahProc[0], INFINITE);

    // Kill the (target) entity if there is one
    if (0 != in_dwIdentKillProc)
    {
        KILL_ENTITY_FROM_IDENT(in_dwIdentKillProc);
    }

    CloseHandle(ahProc[0]);

    // Delete the instances
    DelTree(inp_szFn);

    return true;
}
```

At this point, this embodiment has proven that two assassin process identifiers were specified. This means that the currently executing entity is the first assassin launched. The monitored identifiers will therefore be that of the second assassin entity and the application entity (target). This embodiment will wait for either one to exit; and assumes the target entity will exit when it is finished, in which case the first assassin entity can clean up and itself exit. If, on the other hand, it is the assassin entity that exits, this means that someone or something (a debug process perhaps) has killed it, so the first assassin entity will attempt to terminate the target entity and then delete all the instances of other system entities that it can.

```
ahProc[1] = OpenEntity(ENTITY_ALL_ACCESS,
                FALSE,
                in_dwIdentWaitProc2);

// If we opened handles to both entities, wait for one to exit
if (NULL != ahProc[1])
{
    dwRes = WaitForMultipleObjects(2,          // # of objects to wait for
                    ahProc,      // handles of objs for wait
                    FALSE,       // wait for any 1 obj
                    INFINITE);   // how long to wait

    // If the assassin entity exited, that's an error
    if (WAIT_OBJECT_0 == dwRes)
    {
        // Kill the redirected (target) entity if there is one
```

```
        if (0 != in_dwIdentKillProc)
        {
            KILL_ENTITY_FROM_IDENT(in_dwIdentKillProc);
        }
        CloseHandle(ahProc[0]);
        CloseHandle(ahProc[1]);
        DelTree(inp_szFn);
        DelTree(inp_szFnFk);

        return false;
    }

    CloseHandle(ahProc[1]);
    ahProc[1] = NULL;
}

// Now only the assassin entity is left, so if an additional instance was
//    specified, wait until we can delete it before proceeding
if (NULL != inp_szFnDel)
{
    // Set up instancename
    strcpy(szFnWait, inp_szFn);
    strcat(szFnWait, inp_szFnDel);

    // Wait a while
    for (ii = 0; ii < 180; ii++)
    {
        Sleep(500);

        // Exit the wait if the assassin entity dies or the signal
        //    instance disappears (or we can delete it)
        if ( (!CheckAssassinProc())          ||
            ((-1) == GetInstanceAttributes(szFnWait)) ||
            (DeleteInstance(szFnWait))            )
        {
            break;
        }
    }

    // Kill the instances in our list
    for (ii = 0; ii < INSTANCE_DEL_NUM2; ii++)
    {
        strcpy(szFnDel, inp_szFn);
        strcat(szFnDel, INSTANCE_DEL_LIST2[ii]);
        DeleteInstance(szFnDel);
    }

    // Check if the instance exists
    if ((-1) != GetInstanceAttributes(szFnWait))
    {
        // Wait until either we delete the instance, or the assassin entity is
        //    killed
        while (!DeleteInstance(szFnWait))
        {
```

56

```
                dwRes = WaitForSingleObject(ahProc[0], 250);
                if (WAIT_OBJECT_0 == dwRes)
                {
                    break;
 5              }


        /*
                if (!fTargetInsOpenFailed)
10              {
                    hTargetIns = CreateInstance(szIsDel,
                                GENERIC_WRITE,
                                0,
                                NULL,
15                              OPEN_EXISTING,
                                0, NULL);
                    if (INVALIDENT_HANDLE_VALUE != hTargetIns)
                    {
                        CloseHandle(hTargetIns);
20                  }
                    else
                    {
                        fTargetInsOpenFailed = true;
                    }
25
                }

                // If the instance open failed at least once, try to delete it
                if (fTargetInsOpenFailed)
30              {
                    //DeleteInstance(szTargetIns);
                }
        */
                }
35
        /*
                if (INVALIDENT_HANDLE_VALUE != hTargetIns)
                {
                    CloseHandle(hTargetIns);
40                  hTargetIns = INVALIDENT_HANDLE_VALUE;
                }
        */

                // If the assassin entity was killed, that's an error
45              if (WAIT_OBJECT_0 == dwRes)
                {
                    // Kill the redirected (target) entity if there is one
                    if (0 != in_dwIdentKillProc)
                    {
50                      KILL_ENTITY_FROM_IDENT(in_dwIdentKillProc);
                    }
                    CloseHandle(ahProc[0]);
                    DelTree(inp_szFn);
                    DelTree(inp_szFnFk);
```

57

```
            return false;
          }
        }
5     }


      // Now this invention knows that the target is really done, so clean up and
      // exit
10    CloseHandle(ahProc[0]);
      DelTree(inp_szFn);
      //DelTree(inp_szFnFk);
      // Success
      return true;
15
      } // End WaitAndDeleteInstance()
```

In another embodiment of the present invention, a determination is made by the system as to whether any given process, thread, entity, or access 154 on/of the system is an authorized process or an unauthorized process with respect to access to any of the protected, encrypted, interleaved,or hidden components of the system. As illustrated in **FIG. 25, FIG. 26** establishing such an authorization context and enforcing it involves a series of steps as outlined below. One simple way to illustrate this process is by representing the authorized versus unauthorized entities as "friend or foe", in the form of a list 156. A snapshot of all entities on the system is taken 153 and such a list is established 155. Any entities created subsequently, such as descendant children/entities of the original list entries, are appropriately added to the list 154. When an access occurs, the accessing entity is identified 158 and identity information is compared with the list 159 to determine whether the accessing process is a friend or foe. Access, or denial of access, is issued accordingly 160.

The code example below illustrates the above aspect of the invention as represented in **FIG. 25, FIG. 26**. In the first such example, the identity of an entity is added to the list, and the list is maintained as entity searches reveal new additions:

```
      //
      static VOID OnCreateEntity(
      //
40        DWORD   EntityToken
```

58

```
      )
      {
          Identity_t   entityIdentity;
          Identity_t   DescendantIdentityIdentity = EntityToken ^ s_IdentityObfuscator;
 5        int    ii;
          entityIdentity = (Identity_t)OS_GetCurrentEntityHandle();
          dprintf("Dsrt: OnCreateEntity *** Entity 0x%IX created process 0x%IX \n",
                entityIdentity, DescendantIdentityIdentity);
          // If the entity is in the allowed Identity list add the DescendantIdentity
10        for (ii = 0; ii < MAX_Identity; ii++)
          {
              if (entityIdentity == s_IdentityTable[ii])
              {
                      // If this Identity is already in the Identity array do not add
15                for (ii = 0; ii < MAX_Identity; ii++)
                  {
                      // Found the DescendantIdentity in the table
                      if (s_IdentityTable[ii] == DescendantIdentityIdentity)
                      {
20                        break;
                      }
                  }

                  // Exit outer loop if DescendantIdentity is already in table
25                if ((ii < MAX_Identity) && (s_IdentityTable[ii] == DescendantIdentityIdentity))
                  {
                      break;
                  }

30                // Add a Identity to the array... Any 0 entry will do...
                  for (ii = 0; ii < MAX_Identity; ii++)
                  {
                      if (s_IdentityTable[ii] == 0)
                      {
35                        s_IdentityTable[ii] = DescendantIdentityIdentity;
                          break;
                      }
                  }

40                //if (MAX_Identity == ii)
                  //{
                  // Break out of the outer loop
                  break;

45            } // End if entity is in table

          } // End loop looking for entity in table
          return;
      } // End OnCreateEntity()
50
```

The next code example illustrates the above invention as represented in **FIG. 25, FIG. 26.** In this second such example, the identity of an entity is removed from the list:

```
static VOID OnDestroyEntity(
    DWORD   EntityToken
)
{
    Identity_t   IdentityDescendantIdentity;
    int     ii;


    IdentityDescendantIdentity = EntityToken ^ s_IdentityObfuscator;
    //  Remove this Identity if it is in the list
    for (ii = 0; ii < MAX_Identity; ii++)
    {
        if (IdentityDescendantIdentity == s_IdentityTable[ii])
        {
                s_IdentityTable[ii]);
                s_IdentityTable[ii] = 0;
                break;
        }
    }
    return;
}  //  End OnDestroyEntity()
```

The code example below illustrates mechanisms utilized to verify the identity of an entity and make a decision as to allowing or disallowing access to the entity.

```
// Verify the Identity...
    for (ii = 0; ii < MAX_Identity; ii++)
    {
        if (Identity == s_IdentityTable[ii])
        {
            //if ( (sFunc == FN_OPEN     ) ||
            //    (sFunc == FN_FILEATTRIB) )
            //{
            fIdentityMatch = TRUE;
            break;
        }
    }
```

In another embodiment of this invention, any or all of the above aspects of the invention as illustrated and described above are incorporated into an application, or set of applications, and associated documentation, which are engineered to provide the aforementioned capabilities to digital content creation professionals and other such users. In this manner, digital content that a user desires to protect is provided to an appropriate toolkit as input and the techniques detailed

above are applied to the content. The user is not necessarily exposed to the inner operation of the above processes, nor of the applied inventive techniques. The output of such a toolkit is a protected digital content entity. All types of content are supported and are equally applicable to the principles on the invention, including; audio, video, executable, images, text, documents, e-

5   books, and all other digital content of all types on all platforms as described above. The user of this toolkit may choose to include or exclude any of the inventive components mentioned above as part of the configuration of the tool, but at no time is it necessary for the user to understand in any detail how each component works, or how the individual components of the system interact.

10   In another embodiment, the invention is directed to methods that allow for the electronic delivery of digital content in a fashion that prohibits content modification and duplication by unauthorized persons. The mechanisms detailed herein enable, support and secure the delivery of all forms of electronic content software titles, audio, video, and text/graphic/e-book/e-presentation formats using both hard media and network content delivery models.

15   In one aspect, the product's files are modified, both before an electronic download while the product still resides on the server (or before being copied to the server), and the application files are also modified in the product directory following installation on the customer computer. Hidden data is inserted into these product files, this hidden data incorporating among other identifying data a securely encrypted transaction ID, which may also be modified by a function

20   based on information about the target system's component-specific configuration information. The hidden data may alternately or inclusively comprise a simple numeric index or may also have meaningful content interleaved into itself, as described above in connection with FIG 13. The data may be of any length. These hidden data items are preferably inserted into secret locations within the product files prior to kitting, at the point of purchase.

25   There are multiple authorization models supported by the present invention. One such model is an entirely local model where the digital content may be installed or used from locally read hard media. Another such model is a network model in which there is a series of network transactions. In the latter case, the foregoing may describe a "target Computing System or Device" (the client, in other words) and a "Remote Server" (the server). Note the server may in

30   fact comprise multiple server machines, each performing a set of either unique or redundant

functions. In general, by putting certain key logical components on the remote server they are insulated from being reverse engineered. A good example, detailed elsewhere, is the generation of a system ID based on the configuration of the target system or client. It is desired under some circumstances that this ID value be converted into a usable encryption/decryption key (either

5   symmetric or asymmetric in function). Based on the security concerns just discussed, if this key generation algorithm were on the client system it might be possible to reverse engineer it and compromise it, and thereby compromise the system. By having it be remote on the server in some embodiments of the invention, it effectively becomes a "black box" whose output may of course be intercepted and examined under some circumstances, but whose inner workings cannot

10   be exposed by debugging, disassembly or other compromise of the client system, thus rendering the invention more secure under these circumstances. This client-server utilization of distributed service functions is optimal when possible but may not always be appropriate, as in the case of a CD or other hard media installation or usage of digital content where there is no network connection. Finally, it is important not to view the network versus hard media models as binary

15   opposites or mutually exclusive models, there exists a hybrid model where the installation or distribution and subsequent usage of the digital content is from hard media but certain steps of the authorization process require a network connection. This model is a good compromise whenever possible because it allows certain key algorithm elements to be remote and therefore not susceptible to being attacked by local debug means, and the usage of it requires that the

20   content creator be willing to make an active network connection of some kind (whether broadband, dial-up, or other) a requirement for usage of their content. This set of decisions is entirely a business model based decision, not one limited by technical means. This invention offers optimum flexibility for all such business decisions, local, remote, or hybrid.

As illustrated in the flow diagram of FIG. 27, content is processed and prepared for

25   distribution in the form of a download archive 162, in part using and combining any of the mechanisms illustrated above in connection with FIGs. 3 through 26. The archive is stored, for example at a server, in preparation for a download by a remote user 163. For example, as shown in FIGs. 28 and 29 above, a software or firmware component or tool (embodying technology detailed in FIGs. 3 – 8 above) is deployed to the computing device or system on which the user

30   desires to install or use the desired content (hereinafter referred to the "Target System" or

"Computing Device") and is run. The execution of this tool causes the system's component makeup to be analyzed and examined, and a unique identifying value is generated that represents the examined totality of the system 164. Each of the system's components are examined 165 as desired and selected aspects of each component's properties are considered in producing a

5 unique identifying value for the system 166. For example, as shown in FIG. 30, generation of the identifying value may represent a consideration of component properties information such as the manufacturer, and/or the firmware revision, and/or the serial number, and/or other directly measurable physical properties such as performance, or amount of memory or other storage, or other enumerable hardware features 173 that are aggregated 174 (see FIG. 30) by means of a

10 function, as simple or complex as desired; they may be summed, for example, or mapped to a complex mathematical function yielding a numeric value. This function may look up values in tables or calculate them directly from the input values, or both. Once this value has been arrived at, it is processed 175 into a final system ID value that is used by the system 166 (returning to FIG. 28) in subsequent activity. This value may be used as an element in the creation of a

15 system-unique encyption key, and this key 167 then applied to an encryption process in which the system ID information is encrypted and interleaved with other validation information as shown in FIG. 3 above.

In the embodiments of the present invention that support network distribution of content, depending on the context of the usage of the embodiment, the System ID value may be

20 transferred 168 either to a remote server, a locally executing runtime process, or a locally executing installation process, and further manipulated as shown in FIG. 31 as it is used in conjunction with the Transaction ID 176 and then interleaved and encrypted 177, for example according to the techniques described above with reference to FIGs. 3 through 8. This information is hidden within the content 178 as part of the delivery and optionally any

25 installation process.

With reference to FIG. 29, the identifying data or watermark as created in FIG. 28 is inserted into the archive of the desired digital content product. This occurs on a remote server, usually, but may also occur locally in an installation from CD or other hard media. Note that the remote case is inherently more secure but either case provides an archive with hidden identifying

30 or watermark data. which is hidden and inserted as described in FIG. 31 as in FIGs. 3 - 8. The

identifying watermark data is encrypted and interleaved and hidden in the archive 169. The entire archive, either as one monolithic file or as a collection of smaller files (segmented for faster download performance, more reliable resumption of download, and other network performance and robustness concerns) is encrypted using a key that is made unique for the target

5   computing device system 170 such that is can only be decrypted appropriately on the same system as it was packaged for, as that decryption process will begin with the determination of the system ID of the target as in FIG. 28. The archive is transferred to the target system 171; in the case of a network transaction it is transmitted from the remote server to the target system, while in the case of a local installation, the archived data is provided to the installation process

10   immediately. The recipient of the encrypted archive must of course have the appropriate key with which to decrypt it, and the present invention offers two strategies for providing that key. The recipient process can synthesize its own decryption key as described in FIG. 28 step 167, or can be provided by the remote system or other local process after it consumes the identifying data as provided by FIG. 28 and itself converts it into an appropriate key.

15   Returning to FIG. 27, the components of a digital content product are processed and packaged, for example the standard component contents of a hard media digital product, including executable files, documentation files, image files, and audio files. A standard hard media product may be taken in entirety from a CD release and modified 162 into a more secure downloadable product 163. Some or all of the content may include hidden identifying data

20   inserted into it as shown above with reference to FIG. 31. This hidden identifying data, may optionally contain serialized purchase information related to the individual purchaser at which the Target Computing device is instrumented (step 164 of FIG. 28) and then examined (step 165 of FIG. 28) and a system identifier is created (step 166 of FIG. 28) by examining selected elements of the Target Computing device are identified uniquely and transformed into a unique

25   identifying watermark for later use (steps 173-175 of FIG. 30).

Referring to FIG. 38, these watermarks or hidden data items may be created (the concepts described above in FIGs. 3 through 8 above can also be referred for mechanisms used in the creation, hiding, and later extraction of these data items) may be created using multiple virtual streams as described above, in which, in this example, one such stream 209 represents

30   System ID information containing unique identifying information pertaining to the system, and

64

another such stream contains transaction specific 210 (for example identifiably serialized) information generated by the server at the time of first authentication or download. These streams are interleaved and encrypted 211 (as shown above in FIG. 3). This concept can be used in an Electronic or Network distribution model, and may also be used in a hard media

5   distribution model if the model allows for the Target Computing Device to be at least temporarily connected to a network.

A mechanism of the invention authorizes the execution of product components (also referred to as Content) by providing a service that correlates attributes of the executable product with the system that is the target of the execution of the product. This applies to hard media and

10   to Electronic Software Distribution (ESD). In this aspect of the present invention, the ESD content delivery phase may be authenticated by means of purchase (or other) unique transaction-specific information and/or system specific information. On a server that deploys this protected content, kit components are packaged as large archives or stored as raw data (in the same form as a hard media kit, including optionally, directory structures) and then manufactured on-demand,

15   per-user, per purchase. The final kit is packaged as a collection of encrypted archives, or as a single monolithic archive, securely encrypted, and made usable or installable at the appropriate time by a secure product execution process. This process employs many of the techniques detailed above with reference to FIGs. 3 - 26, and is further strengthened by the client-server authentication process detailed in FIG. 32. In this process, a component capable of determining

20   the unique identifying data is deployed to the Target Computing device. The deployed component finds hidden locations (as in FIGs. 3 through 8 above) within the content or elsewhere on the computing device 179. Once the identifying data is extracted, it is sent via network connection to the server 180 where it is, turning to FIG. 33, received 182, de-interleaved and decrypted 183 (also as in parent filing FIG. 3 through FIG. 8), and its authenticity is verified

25   184 against a database of known authorized content delivery transactions. Both the transaction ID and the system ID are verified 184, and an appropriate response is generated, containing critical content data that is interleaved and encrypted 185 (according to the techniques described above with reference to FIGs. 3 - 8) and then sent to the Target Computing Device 186. Returning to FIG. 32 the response is read from the server 181, the response being a necessary

component to allow for the successful execution of the protected content on the Target

Computing Device as discussed in the foregoing.

In the network installation case, installation or re-installation may be disallowed at any

time by the process illustrated in FIG. 33, in that the server can make authentication decisions

5   based on certain criteria in addition to the overall validity of the system ID and transaction ID

information, for instance total number of authentications or re-authentications, or the frequency

of these events, or other information, can cause the server to choose not to authenticate in the

verification step 184, such that the response data provided can indicate a system decision to

generate a Boolean (i.e. "yes" or "no") state of failure, and/or can contain executable code

10   instructions interleaved with other data (as in parent filing FIG. 3 through FIG. 8) which, when

transmitted at step 186, will cause the recipient process on the Target Computing device to exit,

or to incorrectly decrypt the content and then exit upon a failed content execution sequence.

These server based authentication decisions are also influenced by subsequent transactions in

which the user performs a customer service re-authentication either by phone (verbally) or via

15   the network. Such remote authentication/re-authentication invention methods may be executed

within hard media based products as well.

A mechanism of the invention processes product files in order to make room for larger

quantities of hidden data. These additional spaces for hidden data item are integrated directly

into any desired product files and are optionally pre-filled with filler content that matches the

20   surrounding data to render them difficult to locate. This process modifies each desired product

or content file as follows with reference to FIG. 34. Each desired file is opened and its

filesystem allocation is increased by some amount 187. Internal structure (as required per

filetype) and allocation pointers are modified to match the increased allocation 188. File

contents may be moved within the file as needed 189 as well. All of this available space may

25   optionally be mapped 190, the map hidden, and then later used by another process as described

above with reference to FIGs. 4 - 8.

The process of the present invention may optionally segment the contents of the

download kit such that specific small and critical files or even portions of such files are

segregated from the main kit. The downloaded installation kit is therefore incomplete in small

30   but critical ways. The installation process requires subsequent authenticated reconnections to the

66   .

download host, followed by small and volatile downloads of these critical items. Further this mechanism segments the contents of the installed digital product such that specific critical files and/or portions of such files are segregated and encrypted in a fashion that makes the installed product function properly only on the intended target system. Further, the process of this

5   invention may intentionally leave incomplete certain chosen program elements, the elements to be completed by means of executable information extracted from an authorization process, in some cases by hiding the information within the authentication response. For example the authorization process can provide the system with both numerical encrypted information (keys for further system decryption use) and executable content critical to task completion. In one

10   option, content may have certain sections altered such that key elements are removed and hidden elsewhere in secret locations, for example on the media itself in the case of hard media, on the network in other cases, on other system storage devices in the case for a component already installed on a computer system. Execution requires that these hidden elements are found and replaced in their original locations within the content. These elements are stored in locations that

15   would not be copied easily with either the installer media or the installed product directory.

In a detailed view of such a process, as in FIG. 35, a list of files requiring protection is assembled 191 and each file in turn is processed 192 in that the contents are parsed and certain sections (such as starting point, duration) are identified as critical either by manual selection, or by a file-type and context-aware algorithm (such as a parser or compiler front-end modified for

20   this task), or by a simpler method such as for example by choosing every Nth section M bytes long. Each section is copied to an archive and stored 195 (where it is optionally interleaved and encrypted as described above). Each such section can be identified on an optional map (as per FIGs. 4 – 8 above) 193, and then overwritten with data that is not properly functional 194; for example if the selected file is an executable file, the section may be overwritten with assembly

25   language that exits or that runs in an infinite loop or that causes a severe processing error when run. Upon later use of this protected content, these missing sections must be filled in with the original data.

With reference to FIG. 36, the authentication process for this concept in circumstances that allow the use of a network connection and remote server or servers to assist in the

30   authentication, as each such damaged or modified or incomplete file is read, either as part of a

67

staging process or during the runtime process of the digital content product, access to the specific section of the protected file is redirected through the translocation process as described above with reference to FIGs. 9 - 11 and valid data is substituted for the filler data found in those respective locations. As each such location is accessed or "touched" (this location's filler nature

5   is determined by means of either a Map file as described above, or directly by algorithmic means), the access is blocked in a synchronous manner 196. The blocking protective entity provides a remote server with a system ID 197, using the methods described in FIGs 29 and 30, or extracts the hidden system ID using the methods described in FIGs. 3 – 8, and a request for the missing data item. The remote system or server receives the request and validates the

10   authenticity of the request as described above with reference to FIG. 33, where a valid Transaction ID and a valid System ID are required, and where there may be additional requirements (i.e., number of authentications or re-authentications attempted, frequency, expiry of terms, etc) applied to the generation of the response data. The response data is generated on the server and provided back to the authentication process on the Target Computing System,

15   where it is de-interleaved and decrypted 198 (as in FIGs. 3 - FIG. 8). This response data may contain Boolean flag data indicating the success or failure state of the authentication, and if the authentication fails, the consumption of this protected content can be caused to abort, either immediately, or in a deferred, indirect fashion (see discussion of exit-related processes as disclosed above with reference to FIG. 24, and in FIGs. 44 - 49 below). In addition to such direct

20   methods of communicating a failed authentication within the response data, this process also supports more robust methods for example using the interleaved streams of the response data format to transmit the missing, archived content. One mechanism is the inclusion within the response data of executable data (not from the archive but rather as response to a failed authentication) which causes an exit, an error condition, or which causes communication to

25   another system entity which itself begins a cascading exit process.

The authentication process on the Target Computer System next optionally de-interleaves and decrypts the response data (according to the processes of FIGs. 3 – 8, above) and optionally uses the map data 199 to confirm placement of the data and to optionally determine the next location(s) to block on for subsequent reads. The authorization process then substitutes the filler

30   data (as in FIG. 35 194) with the executable data 200. This may be done as a one-time fix-up of

data during an execution, or may optionally be immediately overwritten after use with more filler data, by means of an event driven synchronized process, such that the corrected data is provided in a volatile, just-in-time fashion for a brief window of time. Access to the data item is then permitted by the system 201.

5   This invention can also be embodied within a variation of the mechanism described above. With reference to FIG. 37, authentication can also performed locally without benefit of a network connection to a server or remote authenticating entity. There are similarities to the logical flow of FIG. 36 above. When a read occurs and the target is one of the files that had previously been processed as in FIG. 35 above, and a location within that file is touched (this

10   location's filler nature is determined by means of either a Map file 204 as described above in FIGs. 3 - 8, or directly by algorithmic means) that had previously been copied to an archive 192 and replaced with filler data 194, the attempt to read this filler data is asynchronously blocked 202, the appropriate System ID information is generated or retrieved 203 (using the methods described in FIG. 29 and FIG. 30 or extracts the hidden system ID using methods described in

15   FIGs. 3 - 8), and the archived data necessary to fill in that allocation is located (see FIGs. 3 - 8) 205 and its target location is correlated with the map 206. The archived data is copied the target location, either as a one-time fix-up of that location, or in a just-in-time fashion where it is replaced with filler data immediately after use by the reading process 207. The read is unblocked 208 and the data is then capable of being read in its new, corrected form.

20   A mechanism of this invention in which there are methods that can detect and discover the presence of classes and instances of software whose effects may be of a compromising nature to the secure system. Such tools whose discovery is desired include those used for software development (known variously as ICES, Debuggers, dump/lift tools, process fixup tools, any and all executing entities on the system at any level of privilege) and which discovery initiates

25   defensive responses (exit, kill intrusion process, etc) when invoked on a system that has been thus instrumented for hacker purposes. In the system and process of the present invention, with reference to FIG. 39, a list of sample patterns is arrived at by examining the in-memory patterns of storage of the tools listed above. Small segments of memory are copied that bear unique information about the target applications and entities. Later, on the target computing device, a

30   protective program is invoked and this list is loaded 212, and the system's memory bounds are

determined 213, for example for all physical and virtual memory, Random Access Memory and other memory including NVRAM, Flash RAM and Virtual Memory Files on read/write media. A starting point within this memory is selected and an ordering for subsequent reads is determined 214. Memory ranges are read into a buffer 215, compared with each item in the list 216, and

5   upon a match an action is taken (such defensive actions as outlined above in FIG. 24, below in FIGs. 44 through 49). After each section is read, in the event there is no match, the memory range is incremented 217 and the process repeats on an iterative basis 218, 219 until all of memory has been scanned. This may be performed at varying levels of priority, and the performance impact of this memory scan upon the system may be throttled by programmatic

10   variables as are the activities detailed above, with reference to FIGs. 17-20.

One such example embodiment is illustrated with the code sample below, in which privileged and unprivileged memory on the target computing device is examined using the methods outlined above:

15

```
// SearchMemory
static BOOL SearchMemory(
DWORD   ProcessIdentifier,
char*   exeName,
BOOL    searchIncremental = FALSE,
int     ringPosition = 0 )
{
    //static INSTANCE  hSEMAPHORE = INVALID_INSTANCE_VALUE;

    BYTE            byBuf[BUFSIZE + MAXLENGTH - 1];
    DWORD           cbRead = 0;
    BOOL            fMemoryRead = FALSE;
    INSTANCE        hSEMAPHORE;
    INSTANCE        hProc = NULL;
    DWORD           ii;
    DWORD           jj;
    BYTE*           pMemBase = (BYTE *)0x00400000;
    char    szSEMAPHORE[32] = "\326\127\107\126\207\362\326\226\067\066"
    char    szMsg[MAX_PATH];

    if (searchIncremental)
        {
        pMemBase = s_pMemBaseStart[ringPosition];
        }
    if (ProcessIdentifier == GetCurrentProcessId())
        {
        return FALSE;
        }

    if (!hProc)
        {
        return FALSE;
        }

    fMemoryRead = TRUE;
```

```
        while (fMemoryRead)
                {
                WaitForSingleObject(hSEMAPHORE, INFINITE);
                fMemoryRead = ReadProcessMemory(hProc,
                        (void *)pMemBase,
                        byBuf,
                        BUFSIZE + MAXLENGTH - 1,
                        &cbRead);
                ReleaseSEMAPHORE(hSEMAPHORE);

            if (!fMemoryRead)
                {
                break;
                }

        // Adjust address for next read
        pMemBase += cbRead;
            if (searchIncremental)
                {
            s_numBytesScanned[ringPosition] += cbRead;
                }

            for (ii = 0; ii < cbRead; ii++)
            {
                for (jj = 0; jj < NUMSIGS; jj++)
                    {
                    if ( MemoryCompare(&(byBuf[ii]),
                        signatures[jj].sig,
                        signatures[jj].length) == 0)
                        {
                            KILLPROCESS(ProcessIdentifier);
                                }
                        CloseHandle(hProc);
                        return TRUE;
                    }
                }
            }
            // Check to see if number of bytes checked so far is greater than MAXINCSCAN
            if ((searchIncremental) && (s_numBytesScanned[ringPosition] > MAXINCSCAN)) {
                s_pMemBaseStart[ringPosition] = pMemBase;
                CloseHandle(hSEMAPHORE);
                CloseHandle(hProc);
                return FALSE;
            }
            }

        if (searchIncremental)
                {
            s_pMemBaseStart[ringPosition] = (BYTE *) 0x400000;
                }
    // Done
        CloseHandle(hSEMAPHORE);
        CloseHandle(hProc);
        return FALSE;

} // End SearchMemory()


static __forceinline BOOL ProcMemScan(
//Return:
//true if we found an instance of an undesireable executable or program running )
{
    BOOL        bStat;
    BOOL        fFound;
    INSTANCE        hSnap;
    PROCESSENTRY32    pe;
    // Init
```

71

```
        pe.dwSize = sizeof(pe);
        fFound = FALSE;

        // Get a snapshot of the current process table
5       hSnap = ProcSnapshot(TH32CS_SNAPPROCESS, 0);
        if ((INSTANCE)(-1) == hSnap)
        {
                //unable to get a snapshot of the current process table
                return FALSE;
10      }

        // Get the 1st process entry
        bStat = Process32First(hSnap, &pe);

15      // Walk through the list, looking for the entry that matches the specified
        //   child process. If we make it all the way through the list without
        //   finding it, declare failure.
        while (bStat && !fFound)
        {
20              // Search the memory space of this process for signatures
                fFound = SearchMemory(pe.th32ProcessID, pe.szExeFile);
            // Get the next process in the snapshot
            bStat = Process32Next(hSnap, &pe);
        }
25      // Done
        CloseHandle(hSnap);
        return fFound;
        } // End ProcMemScan()


30

//PrivilegedProcess Level Scanning - Goes though the PrivProc device list
        if ( (NULL == inp_idxStart)     ||
            (*inp_idxStart == iProgress) )
35      {
            pDdb = VMM_GetDDBList();
            while (NULL != pDdb)
            {//Check for a known instance of an ICE such as SoftIce
                if ( (0x0212 == pDdb->DDB_Req_Device_Number) ||
40                  (0x0252 == pDdb->DDB_Req_Device_Number) ||
                    (0x795A == pDdb->DDB_Req_Device_Number) ||
                    (in_wId == pDdb->DDB_Req_Device_Number) )
                {
                    dwRes |= 0x00000016;
45                  break;
                }
                if (//Search for Monitoring Tools
                    (0 == strncmp( cpy(pOutFilePrivProc, pInFilePrivProc, fLen),
                            (char *)pDdb->DDB_Name, 8)          ) || #endif
50                  (0 == strncmp( cpy(pOutRegPrivProc, pInRegPrivProc, rLen),
                            (char *)pDdb->DDB_Name, 8)          ) || #endif
                    (FALSE                             ) )
                {
55                  dwRes |= 0x00000116;
                    break;
                }

                MemorySet(pOutFilePrivProc,0,sizeof(pOutFilePrivProc));
60              MemorySet(pOutRegPrivProc,0,sizeof(pOutRegPrivProc));
                //Search for debugger tools
                if ( (0 == strncmp( cpy(pOutDebug,pInDebug,dLen),
                            (char *)pDdb->DDB_Name, 8)     ) ||
                    (0x0102 == pDdb->DDB_Req_Device_Number      ) )
65              {
                    dwRes |= 0x00001020;
                    break;
```

72

```
        }
        MemorySet(pOutDebug,0,sizeof(pOutDebug));

        //Find certain hacker tools which are used to conceal ICE tools
5       if ( (0 == pDdb->DDB_V86_API_Proc          ) &&
          (0 == pDdb->DDB_PM_API_Proc              ) &&
          (0 == pDdb->DDB_PrivilegedProcess_Service_Table_Ptr ) &&
          (0 == pDdb->DDB_PrivilegedProcess_Service_Table_Size) &&
          (0 == pDdb->DDB_Flags                )  )
10      {
          dwRes |= 0x00001110;
          break;
        }
        pDdb = (DDB *)pDdb->DDB_Next;
15    }

      if (NULL != outp_mask)
      {
        (*outp_mask) |= (0x00000001 << iProgress);
20    }

      if (dwRes) return dwRes;
    }

25  //PrivilegedProcess Memory Scanning code
    static BOOL ScanMemSegForDw(
      BYTE*     AddressSegment,    // start of segment
      DWORD     in_cbSeg      //  size of segment )
    {
30    DWORD        AddressCheck;
      DWORD        ii;
      DWORD        pos;
      DWORD        pos2;
      DWORD        posMin;
35

      // Make sure section is long enough
      if (in_cbSeg < MIN_SIG_LENGTH)
      {
40      // Section is to short too contain a matching memory pattern signature
        return FALSE;
      }

      // Check for valid address range
45    if (0 == CheckMemoryRange((DWORD)(AddressSegment) >> 12, 1, 0))
      {
        return FALSE;
      }

50
      // Go through the whole segment
      for (ii = 0; ii < (in_cbSeg - 10 + 1); ii++)
      {
        AddressCheck = (DWORD)(&(AddressSegment[ii])) + 10 - 1;
55
        // Check each new page we get to
        if (AddressCheck == (AddressCheck & 0xfffff000))
        {
          if (0 == CheckMemoryRange(AddressCheck >> 12, 1, 0))
60        {
            //Scanner: Address not valid, skipping
            return FALSE;
          }
        }
65
        // Check for "Set Screen Colors" string found in one or more hacker tools
        if (0 == MemoryCompare(&(AddressSegment[ii]), "Set Screen Colors", 10))
```

73

```
                {
                    return TRUE;
                }

5                // Locate load of a specific register
                // Search backwards for a specific instruction
                //this identifies a specific toolset as well
                for (pos2 = pos; pos2 > posMin; pos2--)
                {
10                  if ( ( (*((WORD *)pos2) == 0xb60f)  &&
                        (*((BYTE *)pos2 + 2) == 0x5)  &&
                        (*((BYTE *)pos2 + 6) == 0xc0) )
                    {
                        return TRUE;
15                  }
                }
            }
        } // End walk through segment
        return FALSE;
20  } // End ScanMemSegForDw()

    static DWORD MemScanner(
        int*        inp_idxStart,          // optional ptr to the idx of the PrivilegedProcess
                                           //   to scan
25      DWORD*      outp_mask,
        unsigned long* outp_nPrivProcs,
        char*       outp_szName    // name of PrivilegedProcess containing offending sig )
    {
        DWORD       cbDevData;
30      DWORD       cbTable;
        DWORD       dwRes = 0;
        int         iPrivProcCount = 0;
        int         ii;
        int         jj;
35      DWORD       nPrivProc;
        DEVICEINFO*  pddDynamic = NULL;
        DEVDATA*     pddStatic = NULL;
        //char        szName[30];


40      // Initialize any output args
        if (outp_szName)
        {
            outp_szName[0] = 0;
45      }

        // Search the statically loaded PrivilegedProcesss
        pddStatic = (DEVDATA *)VMM_GetPrivilegedProcessLocationList(&nPrivProc, &cbTable);
        dprintf("Scanner: Static Device Data = 0x%lX   "
50          "PrivilegedProcess Count = %d   "
            "PrivilegedProcess TableSize = %d\n",
            pddStatic, nPrivProc, cbTable);


        iPrivProcCount += nPrivProc;
55
        // Scan the static PrivilegedProcesss if we are doing all PrivilegedProcesss or if one has been
        //   specified and it is in the static list
        if ( (NULL == inp_idxStart) ||
            (*inp_idxStart < nPrivProc) )
60      {
            // Go through all static PrivilegedProcesss
            for (ii = 0; ii < nPrivProc; ii++)
            {
                // If we are doing all PrivilegedProcesss or this is the one specified
65              if ( (NULL == inp_idxStart) ||
                    (ii == *inp_idxStart ) )
                {
```

74

```
        // Scan all of its segments
        for (jj = 0; jj < pddStatic->dd_nSegments; jj++)
        {
            // Skip to the next segment if there's nothing in this one
5           if (0 >= pddStatic->dd_SegData[jj].sd_Size)
            {
                continue;
            }
            if (ScanMemorySegment(pddStatic->dd_SegData[jj].sd_Base,
10                      pddStatic->dd_SegData[jj].sd_Size,
                        outp_szName))
            {
                // Found something, bail
                return 0x10000000;
15          }
        } // End for all segments in curr PrivilegedProcess

        // If a PrivilegedProcess was specified and we just scanned it, the fact that
        //   we made it here means that we didn't find anything
20      if (NULL != inp_idxStart)
        {
            if (NULL != outp_mask)
            {
                (*outp_mask) |= (0x00000001 << (*inp_idxStart));
25          }

            (*inp_idxStart)++;
            return dwRes;
        }
30      }
        // Compute the size of the current DEVDATA struct
        cbDevData = offsetof(DEVDATA, dd_SegData) +
                (pddStatic->dd_nSegments * sizeof(SEGMENTDATA));
        // Move to the next DEVDATA struct
35      pddStatic = (DEVDATA *)(((BYTE *)pddStatic) + cbDevData);
    } // for all static PrivilegedProcesss
}


40  // Now search the dynamically loaded PrivilegedProcesss
    jj = (-1);
    pddDynamic = PrivilegedProcessLDR_GetDeviceList();
    dprintf("Scanner: Dynamic Device Data = 0x%lx\n", pddDynamic);
    // Go through all dynamic PrivilegedProcesss
45  while (pddDynamic)
    {
        // curr idx = nPrivProc + jj
        jj++;
        iPrivProcCount++;
50      if ( (NULL == inp_idxStart)     ||
            (nPrivProc + jj == *inp_idxStart) )
        {
            // If the current PrivilegedProcess has been loaded
            if (0 != pddDynamic->Dl_LoadCount)
55          {
                for (ii = 0; ii < pddDynamic->Dl_ObjCount; ii++)
                {
                    // Skip to the next segment if there's nothing in this one
                    if (0 >= pddDynamic->Dl_ObjInfo[ii].Ol_Size)
60                  {
                        continue;
                    }
                    if (ScanMemorySegment(
                        (BYTE *)pddDynamic->Dl_ObjInfo[ii].Ol_RealAddress,
65                      pddDynamic->Dl_ObjInfo[ii].Ol_Size,
                        outp_szName))
                    {
```

75

```
                    // Found something, bail
                    return 0x20000000;

                }
                //Time_Slice_Sleep(10);
            } // End for all segments in curr PrivilegedProcess
        } // End if the current PrivilegedProcess has been loaded
        // If a PrivilegedProcess was specified and we just scanned it, the fact that
        //   we made it here means that we didn't find anything
        if (NULL != inp_idxStart)
        {
            if (NULL != outp_mask)
            {
                (*outp_mask) |= (0x00000001 << (*inp_idxStart));
            }

            // If the PrivilegedProcess specified was the last one, reset the counter
            //   instead of just incrementing it
            if (NULL == pddDynamic->DI_Next)
            {
                (*inp_idxStart) = 0;
            }
            else
            {
                (*inp_idxStart)++;
            }
            return dwRes;
        }
    } // End if we are doing all PrivilegedProcesss or this is the one specified
    // Move to next PrivilegedProcess in list
    pddDynamic = pddDynamic->DI_Next;
} // End loop through dynamic PrivilegedProcesss
if (NULL != outp_nPrivProcs)
{
    *outp_nPrivProcs = iPrivProcCount;
}
// Done
return dwRes;
} // End MemScanner()
```

FIG. 40 is a flow diagram of a process in accordance with the present invention in which the system determines whether the environment is safe (criteria include absence of some or all software development tools and emulation environments), and then allows the protected title to run. The system's devices are enumerated 220 and the related properties are examined in detail 221 and converted into meaningful numeric values (as well as measurable device performance metrics being converted to similarly meaningful numeric values). These device specific data items are compared to known data for such devices 222 and emulation of devices, wherever possible, is discerned. Upon detecting any such emulated devices, a set of defensive responses are engaged 223, including those detailed in FIG. 24 above and FIGs. 44 - 49 below.

In the code example below, such hardware emulation detection as referenced above is exemplified, in the case of this example, certain status information is compared to known emulated hardware status information:

```
#define k_EMULATED_HARDWARE_1 (0xC245EA77)
#define k_EMULATED_HARDWARE_2 (0xCC7C231E)
#define k_EMULATED_HARDWARE_3 (0xDC3341EC)
if ((k_EMULATED_HARDWARE_1 == info_Ptr->cpu_ID[0])
&& (k_EMULATED_HARDWARE_2 == info_Ptr->cpu_ID[1])
&& (k_EMULATED_HARDWARE_3 == info_Ptr->cpu_ID[2])) {
EMULATED_HARDWARE_TEST_ID_Match = k_EMULATED_HARDWARE_MATCH;
```

Other mechanisms of this aspect of the invention include disabling certain input device (keyboard and mouse, for example) responses as needed.

An invention mechanism that disables specific keystrokes is shown next. Referring to FIG. 41, system memory is searched and mapped and the signature of the section of memory used to handle keyboard operations 224 is found, and certain key definitions within that memory space are found 225 (such as hotkeys which may need to be disabled in order to prevent the foreground invocation of certain debug and software development tools) and are altered so that during the time period that the digital content protective system is 225 is running, the desired keystrokes can be suppressed 226.

A code example of such keystroke suppression follows:

```
// Search through the whole segment (the 6 is for the 6 bytes we
//   are searching for, we don't want to overflow off the end of
//   the segment when we are doing the compare)
for (pos = kbd_driverSegStart;
     pos < kbd_driverSegStart + kbd_driverSegLength - 6 + 1;
     pos++)
{
  if ( (*((DWORD *)pos) == 0x000120cd) &&
       (*((WORD *) pos + 2) == 0x000d) )
  {
    posMin = pos - 100;
    if (posMin < (DWORD)kbd_driverSegStart)
    {
      posMin = (DWORD)kbd_driverSegStart;
    }
    for (pos2 = pos; pos2 > posMin; pos2--)
    {
      if ( (*((WORD *)pos2) == 0xb60f)  &&
           (*((BYTE *)pos2 + 2) == 0x5)  &&
           (*((BYTE *)pos2 + 6) == 0xc0) )
      {
        *( (BYTE *)(&s_HotKeyAddr))     = *((BYTE *) pos2 + 3);
        *(((BYTE *)(&s_HotKeyAddr)) + 1) = *((BYTE *) pos2 + 4);
        *(((BYTE *)(&s_HotKeyAddr)) + 2) = *((BYTE *) pos2 + 5);
```

```
*(((BYTE *)(&s_HotKeyAddr)) + 3) = *((BYTE *) pos2 + 6);
// Disable desired hot key
s_byHotKeyVal = *((BYTE *)s_HotKeyAddr);
*((BYTE *)s_HotKeyAddr) = 0;
// Break out of the backwards search now that we have
//   found what we were looking for
break;
        }
    }
break;
    }
```

FIG. 42 is a flow diagram of a process that controls keyboard access and only allows the keyboard to function as an input device when the target focus for the keyboard is an authorized application or process window. The target computing system's operating system and interfaces are determined by means of system calls 227. The keyboard driver is located in memory, and all memory locations related to keyboard usage are found. The focus of the keyboard is determined 228 and the process identification information associated with the target of that focus is determined. This process identification information (or PID) is compared with a list of PID information maintained by the system (as in FIGs. 25 and 26 above related to determination of identity and authorization on a process by process basis) 229 and a determination is made as to whether to allow or disallow access 230.

FIG. 43 is a flow diagram of a process by which mouse button access is controlled to only allows the mouse buttons to function as an input device when the target focus for the mouse is an authorized application or process window. The target computing system's operating system and interfaces are determined by means of system calls 231. The mouse driver is located in memory, and all memory locations related to mouse button mapping and usage are found. The focus of the mouse is determined 232 and the process identification information associated with the target of that focus is determined. This process identification information (or PID) is compared with a list of PID information maintained by the system (as in FIGs. 25 and 26 above, related to determination of identity and authorization on a process by process basis) 233 and a determination is made as to whether to allow or disallow access 234. ·

In another aspect, in order to defend the system from attack, the system exits upon being compromised or otherwise touched by unauthorized tools or methods. The exit itself may be delayed or deferred to obfuscate the logic behind the exit process. Other cooperating components of this invention (processes, threads, tasks and other logical algorithmic entities)can be

configured such that if one exits for any reasons all the others exit as well. The methods used to determine whether another process has exited include: interprocess communication via standard system synchronization methods; interprocess communication by nonstandard nonobvious synchronization methods including page level memory access using the VM system; page level

5   memory access subverting the VM system by accessing locations physically; implicit exit via polling of a sort; touching areas of memory such that in the event the target process, thread, task or logical construct itself exits, an access violation in the reading process occurs, unhandled, causing it to exit with no decision logic required; event driven exit, where an event occurs that triggers processes to exit; and cross-kill, where the cooperating components of the system kill

10   ALL of each other and THEN themselves upon compromise. These techniques may be used individually or may be utilized in combination to carry out the process of the present invention.

In another aspect, the process of the present invention maintains system security by using message passing and overt system analysis to determine whether other system components have been compromised or have exited. The exit of any entity is sufficient reason for the rest of the

15   system entities to begin exiting in whatever order they determine appropriate. This results in a more or less nondeterministic order of exit for all components, to confuse efforts to understand a cause and effect relationship between actions (such as debugging) and reactions (such as system exit behaviors). As illustrated in FIG. 44, this is an ongoing task and is actually present in various forms in other system entities as desired. All system entities can participate in this

20   process, making them all part of the family of entities referred to as assassin processes above with reference to FIGs. 24 - 26. In step 235, the system sleeps for a specified interval so as not to check too often for other entity status. The sleep duration is a tunable value and may be dynamically altered by the system as desired. Any messages from other entities are read 236 (discussed in detail in FIG. 46 below) and are de-interleaved and decrypted as in FIGs. 3 and 4

25   above. Message content is modified as needed and then re-encrypted and re-interleaved as in FIGs. 3 and 4 and then sent to the next recipient (discussed in FIG. 45 and FIG. 47 below) 237. If either the read or write process indicate the recipient or sending entity has exited, or if a "kill" message is received, or if there is no message waiting after the specified sleep period has ended, this entity can assume the system to be compromised or exiting for other reasons, and itself

30   initiate the exit process. Thus, the entity may kill the process of one or more peers 238, issue a

"kill" message 239, and kill itself after a wait interval determined by a random number generator 240. If there was no reason to kill self or others, then the sleep value is optionally reset according to system tuning inputs as needed 241 and the process begins again 235 at the next interval.

5   The process illustrated in FIG. 45 allows for the writing of messaging information between system entities in a non-obvious fashion, without using system message constructs (which would be the first place an intruder would look for such messages). The messages may be interleaved and encrypted as shown in FIGs. 3 and 4 above. The memory being used by the intended recipient of the message is found and its bounds examined and understood 242. It may

10   or may not be in the memory space of the actual recipient and memory belonging to any third process may be used as needed. The desired message data values are written to a location within the chosen memory space 243. If no such recipient process identification PID or such associated memory is found, the recipient process is assumed to have been compromised or have exited for some other reason 244.

        The process shown in FIG. 46 allows for the reading of messaging information between system entities in a non-obvious fashion, without using system message constructs (which would be the first place an intruder would look for such messages). The messages may be interleaved and encrypted as shown in FIGs. 3 and 4. The memory intended to serve as a recipient repository of the message is found and its bounds examined and understood 245. It may or may not be in

20   the memory space of the actual recipient and memory belonging to any third process may be used as needed. The desired message data values are read from the location within the memory space chosen 246. If no such valid new message, or no such associated memory is found, the sending/writing process is assumed to have been compromised or have exited for some other reason 247.

25   The process shown in FIG. 47 allows for the passage of messaging information between system entities in a non-obvious fashion, with added security, and still without using system message constructs. The messages may be interleaved and encrypted as shown in FIGs. 3 and 4. The system page table is modified such that one physical page is referenced by one or more virtual pages 248. All writes are done to memory locations on such associated virtual pages and

30   never done directly to any virtual pages used directly by the protective system 249, so that no

80

debugger watch points set to these virtual pages would be triggered by such writes. The page table is further examined on each subsequent write and any exit of any system process is noted; in the event of any such exit being noted, the exited process is assumed to have been compromised or to have exited as part of an intended cascading system exit 250.

5   In another aspect of the present invention, system defense related tasks (such as encryption, decryption, message passing, debugger detection via memory scan, etc) are encapsulated within other routines commonly used by the system. For example, it can happen that every time a file is open, this action triggers a defensive routine that scans memory or rewrites memory. In this manner, any and all system activities operate as events that trigger

10   defensive routines, so these routines do not necessarily have to poll or loop as their exclusive method of acting upon the system, and so that removal of these defenses is non-trivial as they can be deeply integrated into every aspect of the system. As in FIG. 48, the digital content protective system functions may be integrated into other system functions inseparably, so that their removal is non-trivial. The standard function of the component 251 is initialized (for

15   example, if the file system's "open" function were modified to contain one of the memory scan functions already described above in FIG. 39). The calls to this interface (in this example, "open") are processed normally 252 while at the same time the protective function is invoked 252 (in this example all or part of the memory scan). Upon completion of the protective function, the standard result of the function 253 is accomplished, and then the standard return status (in

20   this example the standard information about status that the file "open" returns) is returned to the calling process 254 and, as a means for embedded security, the calling process has no way of knowing that any protective function was additionally invoked.

      According to the present invention, each process, thread or task in the system can have a dual, or multiple, role. One is the true functional role of that component (such as decryption), and

25   the other is to participate in the distributed exit functions that are a significant part of the protective function of the system. Such protective exit functions are sometimes referred to as Assassin processes. Any attempt to compromise the system will result in a mass exit of all system components. The distributed nature of this protection across dozens of system tasks results in a very powerful redundant protection model where any attempt to tamper with one part

30   of the system results in a protective response from the rest of the system.

In the process shown in FIG. 49, the digital content protective system functions related to exit deferral and management may be integrated into other system functions inseparably, so that their removal is non-trivial. The standard function of the component 255 is initialized (for example, if the file system's "open" function were modified to contain the exit and messaging related functions described in FIG. 44 through FIG. 47. The calls to this interface (in this example, "open") are processed normally 256 while at the same time the messaging and exit function is invoked 256 (in this example all or part of the memory scan). Upon completion of the messaging and exit function, the standard processing of the ("open" in this example) function is also accomplished 257 (in this example the standard information about status that the file "open" returns) is returned to the calling process 258 and the calling process has no way of knowing that any protective function was additionally invoked.

FIG. 50 illustrates a process by which all strings and other resource elements are encrypted and decrypted by the system in a volatile fashion when used, and then disposed of, such that they cannot be easily searched for within the code either statically or in memory. Product source files can be preprocessed to obscure any searchable strings. Each desired source file is processed in turn as in 259, and the agreed upon search-and-replace delimiters that were placed by the developers are found 260 and the strings between them are read and encrypted and then overwritten into their original locations 261. Each file is fully processed and closed 262 and the next one in turn is opened 259 until all have been so processed.

In the case where strings were encrypted as specified in FIG. 50, they are made usable to the system as needed. With reference to FIG. 50, each such string, as it is read, is passed to a special translation service of the digital content protective system 263 and is decrypted, and returned as a temporary variable 264. The translated value is used as needed 265 (or conversely a value which is desired to be compared to an already encrypted string is passed to the service 263 and the return value in temp storage 264 is then compared as needed). In either case the value is used as needed 265 and then upon completion of usage, the temporary storage is cleared and re-initialized 266.

FIG. 52 is directed to a mechanism by which data values that are critical to the system are read and rewritten by a number of decoy or spoof processes, such that debugger watchpoints on these values, if any, will be triggered excessively and it will be difficult to determine which

82

accesses are decoy and which are valid without much deeper debugging. Desired memory locations are specified by the caller, and found by the protective system 267. Each such location is read 268 and written to 269, in many cases the write being the same value as was present prior (read value 268, write same value 269) to ensure correct operation of any system components

5   requiring that value. Between each such group of rapid reads and writes, the protective process above sleeps 270, 271 a period of time the duration of which is determined by tuning processes as specified in FIG. 17.

       The systems and methods present invention allow system and product code to maintain itself in a difficult-to-modify state even if modification is attempted by a sophisticated debugger,

10   editor or other tool. Key code elements are rewritten in place, in memory, using whatever mode of privilege is required, many times per second (tens, hundreds, tuned to be optimal as needed), at initialization and during execution, so that any attempts to tamper the code will be changed back to the original state. Depending on the nature of the change, the system may also choose to exit as a result of the tampering. For example, in a classic hacker attack, the modification of

15   Import Tables, is defeated in this way. All key code segments are duplicated in an encrypted archive, the archive is hidden (perhaps within files, between files, or outside the file system), and the segments are later read from that archive (some part of the read and decryption occurs in the virtual machine context described elsewhere in the document). Decoy archives and decoy read processes are also established which read from nonencrypted decoy code and write it over the

20   sections, or appear to write it over the sections (writes through the I/O subsystem which are then defeated by tapping into the subsystem and tossing the data away) such that attempts to modify these decoy archives result in no change to the running code. With reference to FIG. 53, product source is preprocessed and delimiters are inserted around critical sections of code. This can be done by for certain code sections by the developers manually, or done by algorithmic means in a

25   less sophisticated selective process. In either event the delimiters are inserted 272, and then the source is compiled 273. When the code is executed 274, a protective entity (either part of each such executing process or independent of the executing process) finds each such marker 275 and overwrites the program data there with the identical program data 276 at a rate of multiple times per second (the frequency is tunable 277 using methods described above with reference to FIG.

83

17). This process is continuous, and occurs in parallel in multiple simultaneous process contexts throughout the system.

A code example of this continuous overwrite process is reproduced below to provide for

5   additional clarity:

```
// Overwrite all methods with a correct copy of the code.
// First we need to decrypt the good code in a temp buffer
// so we have good data to overwrite with

iLengthToEncrypt = validData->validEndAddress  -
            validData->validStartAddress + 1;
iAmountEncrypted = 0;

// Decrypt the buffer
tempBuffer = (BYTE *)malloc(encryptBlockLength);
for (jj = 0; jj < iLengthToEncrypt; jj += 16)
{
    if ( (iLengthToEncrypt == 0)        ||
        (iLengthToEncrypt > iAmountEncrypted) )
    {
        pEncrBlock->DecryptBlock(&(validData->myCode[jj]),
                    tempBuffer);
        iAmountEncrypted += encryptBlockLength;
        MemoryCopy(&(tmpCode[jj]), tempBuffer, encryptBlockLength);
    }
}
// Zero the temp buffer now that we are done with it
memset(tempBuffer, 0, encryptBlockLength);
free(tempBuffer);
// overwrite
MemoryCopy((void *)validData->validStartAddress,
    tmpCode,
    validData->validEndAddress-validData->validStartAddress); #endif
// Zero the full buffer of decrypted code now that we are
//   done with it
memset(tmpCode, 0, sizeof(tmpCode));
break;
}
```

40   The present invention further accommodates certain critical executable components to be processed before shipment to be populated with tens or hundreds of thousands of data values which trigger debugger breakpoints in many debuggers. During normal execution of the title in a non-debug environment, these breakpoints are handled by a null handler and little negative performance impact is achieved. In the debug environment, each breakpoint stops the debugger

45   and requires the intruder to at the least click the mouse and type into the keyboard. A single execution of such a title would require on the order of a hundred thousand mouse-clicks and keyboard presses. The purpose of such is to significantly deter unauthorized debugging, and at the very least to render it as slow and painful as possible. With reference to FIG. 54, the product

source code is pre-processed to insert the desired number of breakpoint values 279. The source is compiled into executable code 280, and the code is run 281 at some later time on the target computer device 281. Upon such execution, each breakpoint in turn 282 is hit. If no debugger is running, then no actual breakpoint handler is invoked, so there is little or no negative system

5   performance impact. In the event an unauthorized debugging tool is in use, each breakpoint results in a functional breakpoint trap execution, and the user will have to (on most debuggers) press a keyboard or mouse key 283 in order to advance the program counter 284. Depending on the sophistication of the debugger, such keypresses may in fact continue to be suppressed by the other protective functions of this system outlined above in FIGs. 41-43 and the system may be at

10   that point hung in an unusable state requiring reboot; an acceptable defensive outcome. Even under the best of circumstances, most debuggers will require the user to press one or more keys or mouse clicks before continuing; in the case where tens of thousands of such breakpoints have been inserted, the burden upon the user exceeds most users limits of patience and the task of debugging is abandoned.

15   In another aspect of the present invention, resistance to tools used to "memory lift" or "memory dump" is achieved by modifying (corrupting) large portions of the code before packaging and storing the original correct portions elsewhere. This modification can take the form of gross and/or subtle corruption, yielding unexecutable code or subtle logical alterations in code that runs. When the code is run in the correct context, a cooperating synchronized system

20   process modifies the code back to the correct executable state but only in a rolling window of context in a just-in-time fashion such that at no time is the entire body of the content correct, just those parts that are required at the current execution time. Once executed these lines of code are re-corrupted. With reference to FIG. 55, the body of product source and or executable code is pre-processed and critical sections of code selected (either manually by skilled developers, or

25   using automated algorithmic methods) and copied to a protected archive, in encrypted form 285. These sections are then overwritten with "incorrect" code 286 which may in fact be subtly incorrect (causing slightly odd behavior of the product) or grossly incorrect (causing the product to exit, or to signal a system wide suicide and exit, or simply to crash). In the event that the product source as the target of such modification, it is then compiled 287 into executable code.

30   At any later time, the protected code is run on a target computing device 288, and upon

85

execution the first "incorrect" code section comes up for execution. Before the incorrect code can be executed, a cooperating system process traps on the program counter 289 attempted read of the "incorrect" area, and the archived code is read and the corrected values are written to that location 290 in a just-in-time fashion. After the corrected code has been executed 291 the section

5   is set back to its incorrect state, such that in the event memory was dumped or lifted during execution, at most one of the multiple incorrect code sections would be correct, and the code therefore would not be useful or fully functional.

The present invention further includes a system and method by which source, object, or executable code is processed to generate variant different versions of executable code, by means

10   of replacement of content with functionally synonymous content. For example in the case of executable content, different assembly language instructions and ordering, that produce the same functional outcome, such that no two such versions share the same fingerprint or the same code-line-number relationship per instruction. This variation is designed to reduce or eliminate the broadly disseminated effectiveness of hacker tutorials and documents that usually depend on

15   specific line-number directions. As illustrated in FIG. 56 by example, each product executable file is opened 292 and parsed 293 such that specific assembly language command constructs are identified and noted. Such constructs are then replaced by synonymous constructs 294 that vary by the type of assembly language command or by the number of assembly language commands required to accomplish the same task, or by both of these factors. After this pass of replacement

20   activities, the entire file is re-ordered by the processing logic 295 wherever possible without causing it to break or altering its logic. This re-ordering may require that assembly language commands additionally be inserted to jump around the file to the new locations and accomplish the correct ordering of tasks as per the original file. This variant product file is written out 296, and the process begins again. Where possible, multiple variants of a given assembly language

25   file are created 297. When all possible variations known to the system have been exhausted, the next product file is opened 298.

Although the invention has been described in language specific to structural features and/or methodological steps, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or steps described. Rather, the specific

30   features and steps are disclosed as preferred forms of implementing the claimed invention.

While this invention has been particularly shown and described with references to preferred embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made herein without departing from the spirit and scope of the invention as defined by the appended claims.

5

10